# Denotational validation of higher-order Bayesian inference

# Denotational Validation of Higher-Order Bayesian Inference

ADAM ŚCIBIOR, University of Cambridge, England, UK and Max Planck Institute for Intelligent Systems, Germany
OHAD KAMMAR, University of Oxford, England, UK
MATTHIJS VÁKÁR, University of Oxford, England, UK
SAM STATON, University of Oxford, England, UK
HONGSEOK YANG, KAIST, South Korea
YUFEI CAI, Universität Tübingen, Germany
KLAUS OSTERMANN, Universität Tübingen, Germany
SEAN K. MOSS, University of Cambridge, England and University of Oxford, England, UK
CHRIS HEUNEN, University of Edinburgh, Scotland, UK
ZOUBIN GHAHRAMANI, University of Cambridge, England, UK and Uber AI Labs, California, USA

We present a modular semantic account of Bayesian inference algorithms for probabilistic programming languages, as used in data science and machine learning. Sophisticated inference algorithms are often explained in terms of composition of smaller parts. However, neither their theoretical justification nor their implementation reflects this modularity. We show how to conceptualise and analyse such inference algorithms as manipulating intermediate representations of probabilistic programs using higher-order functions and inductive types, and their denotational semantics.

Semantic accounts of continuous distributions use measurable spaces. However, our use of higher-order functions presents a substantial technical difficulty: it is impossible to define a measurable space structure over the collection of measurable functions between arbitrary measurable spaces that is compatible with standard operations on those functions, such as function application. We overcome this difficulty using quasi-Borel spaces, a recently proposed mathematical structure that supports both function spaces and continuous distributions.

We define a class of semantic structures for representing probabilistic programs, and semantic validity criteria for transformations of these representations in terms of distribution preservation. We develop a collection of building blocks for composing representations. We use these building blocks to validate common inference algorithms such as Sequential Monte Carlo and Markov Chain Monte Carlo. To emphasize the connection between the semantic manipulation and its traditional measure theoretic origins, we use Kock's synthetic measure theory. We demonstrate its usefulness by proving a quasi-Borel counterpart to the Metropolis-Hastings-Green theorem.

CCS Concepts: • **Mathematics of computing → Metropolis-Hastings algorithm**; **Sequential Monte Carlo methods**; • **Theory of computation → Probabilistic computation**; **Bayesian analysis**; **Denotational semantics**; • **Software and its engineering → Language types**; *Functional languages*; *Interpreters*; Domain specific languages; • **Computing methodologies →** *Machine learning*;

Additional Key Words and Phrases: quasi-Borel spaces, synthetic measure theory, Bayesian inference, applied category theory, commutative monads, Kock integration, initial algebra semantics, sigma-monoids


2475-1421/2018/1-ART60
https://doi.org/10.1145/3158148

**ACM Reference Format:**
Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2018. Denotational Validation of Higher-Order Bayesian Inference. *Proc. ACM Program. Lang.* 2, POPL, Article 60 (January 2018), 29 pages. https://doi.org/10.1145/3158148

## 1 INTRODUCTION

One of the key challenges in Bayesian data analysis is to develop or find an efficient algorithm for estimating the posterior distribution of a probabilistic model with respect to a given data set. This posterior distribution combines prior knowledge encoded in the model and information present in the data set consistently according to the rules of probability theory, but its mathematical definition often involves integration or summation over a large index set and does not yield to an efficient computation strategy immediately. A data scientist typically has to make one of the suboptimal decisions: she has to consult a large body of specialised research in order to pick an algorithm suitable for her model, or to change the model so that it falls into one of those cases with efficient known algorithms for posterior inference, or to face the challenge directly by developing a new algorithm for herself.

Recent probabilistic programming languages aim to resolve this dilemma. They include constructs for describing probability distributions and conditioning, and enable data scientists to express sophisticated probabilistic models as programs. More importantly, they come with the implementation of multiple algorithms for performing posterior inference for models and data sets expressed in the languages. The grand vision is that by using these languages, a data scientist no longer has to worry about the choice or design of such an inference algorithm but focuses on the design of an appropriate model, instead.

In this paper, we provide a denotational validation of inference algorithms for higher-order probabilistic programming languages, such as Church [Goodman et al. 2008], Anglican [Wood et al. 2014] and Venture [Mansinghka et al. 2014]. The correctness of these algorithms is subtle. The early version of the lightweight Metropolis-Hastings algorithm had a bug because of an incorrect acceptance ratio [Wingate et al. 2011]. The correctness often relies on intricate interplay between facts from probability theory and those from programming language theory. Moreover, correctness typically requires stronger results from probability theory than those used for the usual $\mathbb{R}^n$ case in the machine-learning community (e.g., Green's measure-theoretic justification of Markov Chain Monte Carlo rather than the usual one for $\mathbb{R}^n$ based on density functions).

Our starting point is the body of existing results on validating inference algorithms for probabilistic programs [Borgström et al. 2016; Hur et al. 2015]. Those earlier results tend to be based on operational semantics, and often (not always) focus on first-order programs. By working in a modular way with monads, denotational semantics and higher-order functions, we are able to validate sophisticated inference algorithms, such as resample-move Sequential Monte Carlo [Doucet and Johansen 2011], that are complex yet modular, being composed of smaller reusable components, by combining our semantic analysis of these components.

The probabilistic programming language considered in the paper includes continuous distributions, which means that semantic accounts of them or their inference algorithms need to use measure theory and Lebesgue integration. Meanwhile, our semantic account uses a meta-language with higher-order functions for specifying and interpreting intermediate representations of probabilistic programs that are manipulated by components of inference algorithms. Such higher-order functions let us achieve modularity and handle higher-order functions in the target probabilistic programming language. These two features cause a tension because it is impossible to define a measurable space structure over the collection of measurable functions between arbitrary measurable

spaces that is compatible with standard operations on those functions, such as function application. We resolve the tension using quasi-Borel spaces [Heunen et al. 2017], a recently proposed mathematical structure that supports both function spaces and continuous distributions.

We define a semantic class of structures for various intermediate representations of probabilistic programs, and semantic validity criteria for transformations of these representations in terms of distribution preservation. We develop a collection of building blocks for composing representations. We use these building blocks to validate common inference algorithms such as Sequential Monte Carlo and Markov Chain Monte Carlo. To emphasize the connection between the semantic manipulation and its traditional measure theoretic origins, we use Kock's synthetic measure theory. We demonstrate its usefulness by proving a quasi-Borel counterpart to the Metropolis-Hastings-Green theorem.

To ease the presentation, we proceed in two steps. First, we present our development in the discrete setting, where the set-theoretic account is simpler and more accessible. Then, after developing an appropriate mathematical toolbox, we transfer this account to the continuous case. Inference in the continuous setting, while conceptually very similar to the discrete case, is inseparable from our development. The semantic foundation for continuous distributions over higher-order functions has been very problematic in the past. The fact that our approach *does* generalise to the continuous case, and does so smoothly, is one of our significant contributions, only brought about through the careful combination of quasi-Borel spaces, synthetic measure theory, the metalanguage, and the inference building blocks.

The rest of the paper is structured as follows. Sec. 2 presents a core calculus, our metalanguage, with its type system and set-theoretic denotational semantics. Sec. 3 presents the core ideas of our development in a simpler discrete set-theoretic setting. Sec. 4 reviews the mathematical concepts required for dealing with continuous distributions. Sec. 5 presents representations and transformations for continuous distributions. Sec. 6 decomposes the common Sequential Monte Carlo inference algorithm into simpler inference representations and transformations. Sec. 7 similarly decomposes the general Trace Markov Chain Monte Carlo algorithm. Sec. 8 concludes. Basic results in synthetic measure theory are listed for the reader's convenience in Appendix A.

## 2 THE CORE CALCULUS

We use a variant of the simply-typed $\lambda$-calculus with sums and inductive types, base types and constructors, primitives, and primitive recursion, but without effects. We also use monad-like constructs in the spirit of Moggi's computational $\lambda$-calculus [1989]. The core calculus is very simple, and at places we need an inherently semantic treatment, which the core calculus alone cannot express. In those cases, we resort directly to the semantic structures, sets or spaces. However, the calculus still serves a very important purpose: every type and function expressed in it denote well-formed objects and well-formed morphisms. In the continuous case, using this calculus yields correct-by-construction quasi-Borel spaces and their morphisms, avoiding a tedious and error-prone manual verification. Using the core calculus also brings our theoretical development closer to potential implementations in functional languages.

### 2.1 Syntax

Fig. 1 (top) presents the types of our core calculus. To support inductive types, we include type variables, taken from a countable set ranged over by $\alpha, \beta, \gamma, \ldots$. Our kind system will later ensure these type variables are *strictly positive*: they can only appear free covariantly — to the right of a function type. Variant types use constructor labels taken from a countable set ranged over by $\ell, \ell_1, \ell_2, \ldots$. Variant types are in fact partial functions with a finite domain from the set of constructor labels to the set of types. When $\sigma$ is a variant type, we write $(\ell\ \tau) \in \sigma$ for the assertion

$$
\begin{array}{lll}
\tau, \sigma, \rho & ::= & \text{types} \\
& \alpha & \text{positive variable} \\
& \mid \{\ell_1\, \tau_1 \mid \ldots \mid \ell_n\, \tau_n\} & \text{variant} \\
& \mid \mathbf{1} \mid \tau * \sigma & \text{finite product} \\
& \mid \mu\alpha.\tau & \text{inductive type}
\end{array}
\qquad
\begin{array}{ll}
\mid \tau \to \sigma & \text{function} \\
\mid \mathsf{A} & \text{base} \\
\mid \mathsf{F}\,\tau & \text{base constructors} \\
\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n & \text{variable contexts}
\end{array}
$$

$$
\begin{array}{lll}
t, s, r & ::= & \text{terms} \\
& x & \text{variable} \\
& \mid \tau.\ell\, t & \text{variant constructor} \\
& \mid () \mid (t, s) & \text{nullary and binary tuples} \\
& \mid \tau.\mathbf{roll} & \text{iso-inductive constructor} \\
& \mid \lambda x : \tau.t & \text{function abstraction} \\
& \mid \mathbf{match}\, t & \text{pattern matching: variants} \\
& \quad \mathbf{with}\, \{\ell_1\, x_1 \to s_1 \mid \cdots \mid \ell_n\, x_n \to s_n\} &
\end{array}
\qquad
\begin{array}{ll}
\mid \mathbf{match}\, t & \text{binary products} \\
\quad \mathbf{with}\, (x, y) \to s & \\
\mid \mathbf{match}\, t & \text{inductive types} \\
\quad \mathbf{with\ roll}\, x \to s & \\
\mid \tau.\mathbf{fold}\, t & \text{inductive recursion} \\
\mid t\, s & \text{function application} \\
\mid \varphi & \text{primitive}
\end{array}
$$

Fig. 1. Core calculus types (top) and terms (bottom)

that $\sigma$ assigns the type $\tau$ to $\ell$. We include the standard unit type, binary products, and function types. We include unary uninterpreted base types and constructors. While we use a list syntax for variable contexts $\Gamma$, they are in fact partial functions with a finite domain from the countable set of variables, ranged over by $x, y, z, \ldots$, to the set of types.

We desugar stand-alone labels in a variant type $\{\cdots \mid \ell \mid \cdots\}$ to the unit type $\{\cdots \mid \ell\,() \mid \cdots\}$. We also desugar seemingly-recursive type declarations $\tau := \sigma[\alpha \mapsto \tau]$ to $\tau := \mu\alpha.\sigma$.

*Example 2.1.* The type of booleans is given by $\mathsf{bool} := \{\mathsf{True} \mid \mathsf{False}\}$. The type of natural numbers is given by $\mathbb{N} := \{\mathsf{Zero} \mid \mathsf{Succ}\,\mathbb{N}\}$ desugaring to $\mathbb{N} := \mu\alpha.\{\mathsf{Zero} \mid \mathsf{Succ}\,\alpha\}$. The type of $\alpha$-lists is given by $\mathsf{List}\,\alpha := \{\mathsf{Nil} \mid \mathsf{Cons}\,\alpha * \mathsf{List}\,\alpha\}$, desugaring to $\mathsf{List}\,\alpha := \mu\beta.\{\mathsf{Nil} \mid \mathsf{Cons}\,\alpha * \beta\}$.

Base types and constructors allow us to include semantic type declarations into our calculus. For example, we will always include the following base types:

- $\mathbb{I}$ : unit interval $[0, 1]$;
- $\overline{\mathbb{R}}$: extended real line $[-\infty, \infty]$;
- $\overline{\mathbb{R}}_+$: non-negative extended reals $[0, \infty]$.
- $\mathbb{R}$: real line $(-\infty, \infty)$;
- $\mathbb{R}_+$: non-negative reals $[0, \infty)$; and

In addition, once we define a type constructor such as $\mathsf{List}\,\alpha$, we will later reuse it as a base type constructor $\mathsf{List}\,\tau$, effectively working in an extended calculus. Thus we are working with a family of calculi, extending the base signature with each type definition in our development.

Fig. 1 (bottom) presents the terms in our core calculus. Variant constructor terms $\tau.\ell\, t$ are annotated with their variant type $\tau$ to avoid label clashes. The tupling constructors are standard. We use *iso-inductive* types: construction of inductive types requires an explicit rolling of the inductive definition such as $\mathbb{N}.\mathbf{roll}\,(\mathsf{Zero}())$. Variable binding in function abstraction is *intrinsically typed* in standard Church-style. We include standard pattern matching constructs for variants, binary products, and inductive types. We include a structural recursion construct $\tau.\mathbf{fold}$ for every inductive type $\tau$. Function application is standard, as is the inclusion of primitives.

To ease the construction of terms, we use the standard syntactic sugar (e.g. $\mathbf{let}\, x = t\, \mathbf{in}\, s$ for $(\lambda x.\, t)s$, $\mathbf{if\ then\ else}$ for pattern matching booleans), informally elide types from the terms, elide $\mathbf{roll}$ing/unrolling inductive types, and informally use nested pattern matching.

*Example 2.2.* For $\mathsf{List}\,\tau = \mu\alpha.\{\mathsf{Nil} \mid \mathsf{Cons}\,\tau * \alpha\}$, we can express standard list manipulation:

$$x :: x_s = \mathsf{Cons}(x, x_s) \qquad \mathsf{foldr}\, a\, f = \mathsf{List}\,\tau.\mathbf{fold}\,\lambda\{\,\mathsf{Nil} \to a \mid \mathsf{Cons}\,(x, b) \to f(x, b)\}$$

$$x_s +\!\!+ y_s = \mathsf{foldr}\, y_s\, (::)\, x_s \qquad \mathsf{map}\, f\, x_s = \mathsf{foldr}\,[\,]\,(\lambda\{\,(y, y_s) \to (f(y), y_s)\})$$

$$\frac{}{\Delta \vdash_{\mathrm{k}} \alpha : \mathrm{type}}(\alpha \in \Delta) \qquad \frac{\text{for all } 1 \le i \le n\text{: } \Delta \vdash_{\mathrm{k}} \tau_i : \mathrm{type}}{\Delta \vdash_{\mathrm{k}} \{\ell_1\, \tau_1 \mid \ldots \mid \ell_{\mathrm{n}}\, \tau_n\} : \mathrm{type}} \qquad \frac{}{\Delta \vdash_{\mathrm{k}} \mathbf{1} : \mathrm{type}}$$

$$\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{type} \qquad \Delta \vdash_{\mathrm{k}} \sigma : \mathrm{type}}{\Delta \vdash_{\mathrm{k}} \tau * \sigma : \mathrm{type}} \qquad \frac{\Delta, \alpha \vdash_{\mathrm{k}} \tau : \mathrm{type}}{\Delta \vdash_{\mathrm{k}} \mu\alpha.\tau : \mathrm{type}} \qquad \frac{\vdash_{\mathrm{k}} \tau : \mathrm{type} \qquad \Delta \vdash_{\mathrm{k}} \sigma : \mathrm{type}}{\Delta \vdash_{\mathrm{k}} \tau \to \sigma : \mathrm{type}}$$

$$\frac{}{\Delta \vdash_{\mathrm{k}} \mathrm{A} : \mathrm{type}} \qquad \frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{type}}{\Delta \vdash_{\mathrm{k}} \mathrm{F}\,\tau : \mathrm{type}} \qquad \frac{\text{for all } (x : \tau) \in \Gamma\text{: } \vdash_{\mathrm{k}} \tau : \mathrm{type}}{\vdash_{\mathrm{k}} \Gamma : \mathrm{context}}$$

Fig. 2. Core calculus kind system

$$\frac{}{\Gamma \vdash x : \tau}((x : \tau) \in \Gamma) \qquad \frac{\Gamma \vdash t : \tau_i}{\Gamma \vdash \tau.\ell_i\, t : \tau}((\ell_{\mathrm{i}}\, \tau_i) \in \tau) \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash (t, s) : \tau * \sigma} \qquad \frac{}{\Gamma \vdash \tau.\mathbf{roll} : (\sigma[\alpha \mapsto \tau]) \to \tau}(\tau = \mu\alpha.\sigma) \qquad \frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau.t : \tau \to \sigma}$$

$$\frac{\Gamma \vdash t : \{\ell_1\, \tau_1 \mid \ldots \mid \ell_{\mathrm{n}}\, \tau_n\} \qquad \text{for each } 1 \le i \le n\text{: } \Gamma, x_i : \tau_i \vdash s_i : \tau}{\Gamma \vdash \mathbf{match}\ t\ \mathbf{with}\ \{\ell_1\, x_1 \to s_1 \mid \cdots \mid \ell_{\mathrm{n}}\, x_n \to s_n\} : \tau}$$

$$\frac{\Gamma \vdash t : \sigma * \rho \qquad \Gamma, x : \sigma, y : \rho \vdash s : \tau}{\Gamma \vdash \mathbf{match}\ t\ \mathbf{with}\ (x, y) \to s : \tau} \qquad \frac{\Gamma \vdash t : \mu\alpha.\sigma \qquad \Gamma, x : \sigma[\alpha \mapsto \mu\alpha.\sigma] \vdash s : \tau}{\Gamma \vdash \mathbf{match}\ t\ \mathbf{with}\ \mathbf{roll}\ x \to s : \tau}$$

$$\frac{\Gamma \vdash t : (\sigma[\alpha \mapsto \rho]) \to \rho}{\Gamma \vdash \tau.\mathbf{fold}\ t : \tau \to \rho}(\tau = \mu\alpha.\sigma) \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t\, s : \tau} \qquad \frac{}{\Gamma \vdash \varphi : \tau_\varphi}$$

Fig. 3. Core calculus type system

where we abbreviate $[a_1, \ldots, a_n]$ to $\mathrm{Cons}\,(a_1, \ldots, \mathrm{Cons}\,(a_n, \mathrm{Nil})\ldots)$.

## 2.2 Type System

To ensure the well-formedness of types, which involve type variables, we use a simple kind system, presented in Fig. 2. Each kinding judgement $\Delta \vdash_{\mathrm{k}} \tau : \mathrm{type}$ asserts that a given type $\tau$ is well-formed in the *type variable context* $\Delta$, which is finite set of type variables.

The kinding judgements are standard. All type variables must be bound by the enclosing context, or by an inductive type binder. The contravariant position in the function type $\tau \to \sigma$ must contain a *closed* type, ensuring that free type variables can only appear in strictly positive positions. Variable contexts $\Gamma$ must only assign closed types.

*Example 2.3.* The types from Ex. 2.1 are well-kinded: $\vdash_{\mathrm{k}} \mathrm{bool}, \mathbb{N}, \mathrm{List}\ \alpha : \mathrm{type}$.

We define capture avoiding substitution of types for type variables in the standard way, which obeys the usual structural properties. Henceforth we consider only well-formed types in context, leaving the context implicit wherever possible, and gloss over issues of alpha-convertibility of bound type variables.

To type terms, we assume each primitive $\varphi$ has a well-formed type $\vdash_{\mathrm{k}} \tau_\varphi : \mathrm{type}$ associated with it. Fig. 3 presents the resulting type system. Each typing judgement $\Gamma \vdash t : \tau$ asserts that a given term $t$ is well-typed with the well-formed closed type $\vdash_{\mathrm{k}} \tau : \mathrm{type}$ in the variable context $\vdash_{\mathrm{k}} \Gamma : \mathrm{context}$.

The rules are standard. By design, every term has at most one type in a given context.

*Example 2.4.* Once desugared, the list manipulation terms from Ex. 2.2 have types:

$$(::) : \tau * \mathsf{List}\,\tau \to \mathsf{List}\,\tau \qquad \mathsf{foldr} : \sigma * (\tau * \sigma \to \sigma) * \mathsf{List}\,\tau \to \sigma$$
$$\mathsf{map} : (\tau \to \sigma) \to (\mathsf{List}\,\tau \to \mathsf{List}\,\sigma) \qquad (+\!\!+) : (\mathsf{List}\,\tau) * (\mathsf{List}\,\tau) \to \mathsf{List}\,\tau$$

### 2.3 Primitive Recursion

As is well-known [Geuvers and Poll 2007; Hutton 1999], structural recursion on inductive types allows us to express primitive recursion. By 'primitive recursion', we mean recursing through values of an inductive type $\mu\alpha.\sigma$ using a term of the form: $\Gamma, k : \sigma[\alpha \mapsto (\mu\alpha.\sigma) * \rho] \vdash t : \rho$ with the intention that $t$ can use either arbitrary (total) processing on the sub-structures of its input $k$, or make a primitive recursive call to itself with a sub-structure. In order to desugar such a term into a function of type $\tau * (\mu\alpha.\sigma) \to \rho$, we use terms of the following type, defined by induction on types:

$$\pi_{\alpha.\sigma,\rho} : \sigma[\alpha \mapsto (\mu\alpha.\sigma) * \rho] \to \sigma[\alpha \mapsto \mu\alpha.\sigma]$$

and interpret the primitive recursive declaration $t$ embodied by:

$$\Gamma, x : \mu\alpha.\sigma \vdash \mathbf{match}\,(\mu\alpha.\sigma).\mathbf{fold}\left(\lambda k : \sigma[\alpha \mapsto (\mu\alpha.\sigma) * \rho].\,(\mathbf{roll}\,\pi_{\alpha.\sigma,\rho}k, t)\right) x \\ \mathbf{with}\,(\_, r) \to r : \sigma$$

This translation is global in nature: the structure of the term $\pi$ depends on the type of $t$. Thus, it does not constitute a *macro* translation [Felleisen 1991]. With this point in mind, we will allow ourselves to use primitive recursive definitions.

*Example 2.5.* We define a function $\mathsf{aggr} : \mathsf{List}(\mathbb{R}_+ * X) \to \mathsf{List}(\mathbb{R}_+ * X)$ which takes a list of weighted values and aggregates all the weights based on their values. We make use of the auxiliary function $\mathsf{add} : (\mathbb{R}_+ * X) * \mathsf{List}(\mathbb{R}_+ * X) \to \mathsf{List}(\mathbb{R}_+ * X)$, which adds a weighted value to an already aggregated list. We define add by primitive recursion:

$$\begin{array}{lll} \mathsf{add}((s,a), x_s) \coloneqq \mathbf{match}\, x_s\, \mathbf{with}\, \{[\,] & \to [(s,a)] & \text{-- new entry} \\ \quad (r,x) :: x_s & \to \mathbf{if}\quad x = a & \\ & \quad \mathbf{then}\,(s+r,a) :: x_s & \text{-- accumulate} \\ & \quad \mathbf{else}\;\; (r,x) :: \mathsf{add}((s,a), x_s)\} & \text{-- recurse} \end{array}$$

and set $\mathsf{aggr} \coloneqq \mathsf{foldr}\,[\,]\,\mathsf{add}$. This example makes use of an equality predicate between $X$ elements, restricting its applicability.

### 2.4 Denotational Semantics

We give a set-theoretic semantics to the calculus. In such set-theoretic semantics, types-in-context $\Delta \vdash_{\mathrm{k}} \tau$ : type are interpreted as functors $[\![\tau]\!] : \mathbf{Set}^{\Delta} \to \mathbf{Set}$, i.e., $[\![\tau]\!]$ assigns a set $[\![\tau]\!]\,(X_\alpha)_{\alpha\in\Delta}$ for every $\Delta$-indexed tuple of sets, and a function

$$[\![\tau]\!]\,(f_\alpha : X_\alpha \to Y_\alpha)_{\alpha\in\Delta} : [\![\tau]\!]\,(X_\alpha) \to [\![\tau]\!]\,(Y_\alpha)$$

for every $\Delta$-indexed tuple of functions between the sets with corresponding index, and this assignment preserves composition and identities.

In order to interpret iso-inductive types $\mu\alpha.\tau$, we need canonical isomorphisms between the sets $[\![\tau]\!]\,([\![\mu\alpha.\tau]\!]) \cong [\![\mu\alpha.\tau]\!]$. We will do this in a standard way, by interpreting $[\![\mu\alpha.\tau]\!]$ as the initial algebra for the functor $[\![\tau]\!] : [\mathbf{Set}^{\Delta} \to \mathbf{Set}] \to [\mathbf{Set}^{\Delta} \to \mathbf{Set}]$. This means that for every functor $A : \mathbf{Set}^{\Delta} \to \mathbf{Set}$ with a natural family of functions $\{a_X : ([\![\tau]\!]\,A)(X) \to A(X)\}_{X\in\mathbf{Set}^{\Delta}}$, there is a canonical natural family of functions $\{\mathsf{fold}_X : [\![\mu\alpha.\tau]\!]\,(X) \to A(X)\}_{X\in\mathbf{Set}^{\Delta}}$.

A technical requirement is needed to ensure that this initial algebra exists: we fix a regular cardinal $\kappa$, and demand that each type denotes a $\kappa$-*ranked* functor (ranked functor for short), that is,

$$[\![\alpha]\!]\, d \coloneqq d(\alpha) \qquad \left[\!\!\left[\{\ell_1\, \tau_1 \mid \ldots \mid \ell_n\, \tau_n\}\right]\!\!\right] d \coloneqq \sum_{i=1}^{n} [\![\tau_i]\!]\, d \qquad [\![\mathbf{1}]\!]\, d \coloneqq \mathbb{1}$$

$$[\![\tau * \sigma]\!]\, d \coloneqq ([\![\tau]\!]\, d) \times ([\![\tau]\!]\, d) \qquad [\![\mathrm{A}]\!]\, d \coloneqq \mathcal{B}\,[\![\mathrm{A}]\!]$$

$$[\![\mathsf{F}\,\tau]\!]\, d \coloneqq \mathcal{B}\,[\![\mathsf{F}]\!]\,([\![\tau]\!]\, d) \qquad [\![\mu\alpha.\tau]\!]\, d \coloneqq \mu X.\, [\![\tau]\!]\, d[\alpha \mapsto X] \qquad [\![\tau \to \sigma]\!]\, d \coloneqq ([\![\sigma]\!]\, d)^{[\![\tau]\!]\,()}$$

Fig. 4. Core calculus type-level semantics

that it denotes a functor that preserves $\kappa$-filtered colimits[1]. The $\kappa$-ranked functors are closed under composition, products, sums, and initial algebras. Initial algebras for $\kappa$-ranked functors on locally presentable categories always exist, because they can be built in an iterative way by transfinite induction (see e.g. [Kelly 1980]).

*2.4.1 Set-Theoretic Interpretation.* To interpret types, we assume a given interpretation $\mathcal{B}\,[\![-]\!]$ of the base types A as sets $\mathcal{B}\,[\![\mathrm{A}]\!]$ and of base type constructors F as ranked functors $\mathcal{B}\,[\![\mathsf{F}]\!] : \mathbf{Set} \to \mathbf{Set}$. We then interpret each well-formed type in context $\Delta \vdash_{\mathrm{k}} \tau : \mathrm{type}$ as a ranked functor $[\![\tau]\!] : \mathbf{Set}^{\Delta} \to \mathbf{Set}$, as depicted in Fig. 4.

In this definition, the parameter $d$ may be either a tuple of sets or functions. When interpreting type variables, we write $d(\alpha)$ for the $\alpha$-indexed component of $d$. The interpretation of simple types uses disjoint unions, singletons, finite products, and exponentials, i.e. the bi-cartesian closed structure of **Set**. We interpret inductive types $[\![\mu\alpha.\tau]\!]\, d$ using the initial algebra for the ranked functor $\lambda X.\, [\![\tau]\!]\, d[\alpha \mapsto X] : \mathbf{Set} \to \mathbf{Set}$. In the semantics of the function type $\tau \to \sigma$, the exponential makes no use of the functor's arguments, and relies on the fact that all type variables are strictly positive. We use the given interpretation of base types and type constructors to interpret them.

Lemma 2.6. *The semantics of types is well-defined: every well-formed type* $\Delta \vdash_{\mathrm{k}} \tau : \mathrm{type}$ *denotes a ranked functor* $[\![\tau]\!] : \mathbf{Set}^{\Delta} \to \mathbf{Set}$. *In particular, every closed type denotes a set.*

The proof is by induction on the kinding judgements, using well-known properties of **Set**.

We will always interpret the base types $\mathbb{I}$, $\mathbb{R}$, etc. by the sets they represent.

*Example 2.7.* We calculate the denotations of the types from Ex. 2.1. Booleans denote a two-element set $[\![\mathrm{bool}]\!] = \{\mathrm{False}, \mathrm{True}\}$, and the natural numbers denote the set of natural numbers $[\![\mathbb{N}]\!] = \mathbb{N}$. By Lemma 2.6, $[\![\mathrm{List}]\!]$ denotes a ranked functor $\mathrm{List} : \mathbf{Set} \to \mathbf{Set}$, and this functor is given by the set of sequences of $X$-elements $\mathrm{List}\, X \coloneqq \bigcup_{n \in \mathbb{N}} X^n$.

Beyond establishing the well-definedness of the semantic interpretation, Lemma 2.6 equips us with syntactic means to define ranked functors. Once defined, we can add these functors to our collection of base types (in an extended instance of the core calculus). In the sequel, we will often restrict a given ranked functor $F : \mathbf{Set} \to \mathbf{Set}$ by specifying a subset $GX \subseteq FX$. Doing so is analogous to imposing an *invariant* on a datatype. The subsets $GX$ form a *subfunctor* $G \subseteq F$ precisely if they are closed under the functorial action of $F$, i.e., for every function $f : X \to Y$ and $a \in GX$, $Ff(a) \in GY$.

Lemma 2.8. *Subfunctors of ranked functors over* **Set** *are ranked.*

We can prove this lemma directly, but it also follows from a higher-level argument using the commutation of finite limits and $\kappa$-directed colimits in **Set**.

[1]We do not use simpler classes of functors, such as *polynomial functors* or *containers*, as they are not closed under subfunctors, given by subsets in the discrete case and subspaces in the continuous case, which we need in the sequel.

**instance** *Monad* (List) **where**
  **return** $x = [x]$
  $x_s \gg\!= f = \text{foldr}\,[\,]\ \big(\lambda(x, y_s).\ f(x) +\!\!+ y_s\big)\ x_s$

(a) Declaring monadic interfaces

| Sugar | Elaboration |
|---|---|
| • $x \leftarrow t; s$ | $t \gg\!= \lambda x.\, s$ |
| • **return** $t$ | $\text{return}^{\underline{T}}\, t$ |
| • $t; s$ | $\_ \leftarrow t; s$ |

(b) Haskell's do-notation

Fig. 5. Monadic programming notation

### 2.5 Monadic Programming

In the sequel, we will be working with types that support a monadic programming style. More precisely, a *monadic interface* $\underline{T}$ consists of a triple $\underline{T} = (T, \text{return}^{\underline{T}}, \gg\!=^{\underline{T}})$ where: $T$ assigns to each set $X$ a set $TX$; $\text{return}^{\underline{T}}$ assigns to each set $X$ a function $\text{return}^{\underline{T}}_X : X \to TX$; and $\gg\!=^{\underline{T}}$ assigns to each pair of sets $X$ and $Y$ a function $\gg\!=^{\underline{T}}_{X,Y} : TX \times (TY)^X \to TY$. We borrow Haskell's type-class syntax to define such interfaces. As an example, Fig. 5a defines a monadic interface over List.

Each such monadic interface $\underline{T}$ allows us to use standard do-notation summarised in Fig. 5b. Though simple in principle, we must take care when treating this notation as syntactic sugar, as choosing the appropriate function $\text{return}_X$ or $\gg\!=_{X,Y}$ at each desugaring step must take typing information into account. When we use do-notation in the sequel, we ensure that such choices can be disambiguated. Finally, we will delimit our use of do-notation to within a *do-block* $\underline{T}$.**do** $\{\ldots\}$, omitting the monadic interface $\underline{T}$ or the entire delimiter when either is clear from the context.

Importantly, we do not insist that a monadic interface satisfies the monad associativity and unit laws: $(\text{return}\, x) \gg\!= f = f(x)$, $a \gg\!= \text{return} = a$, and $(a \gg\!= f) \gg\!= g = a \gg\!= (\lambda x.\, (f\ x \gg\!= g))$.

## 3 DISCRETE INFERENCE

We can now lay-out the core ideas in the simpler, set-theoretic case: a semantic structure for higher-order (discrete) probabilistic programs, intermediate representations of these programs for the purpose of inference, valid transformations between these representations, and modular building blocks for creating new representations and transformations from existing ones. For simplicity, we consider representations and transformations from simple rather naive inference algorithms only in this section. In Sec. 6 and Sec. 7, we show how the core ideas here apply to advanced algorithms when aided with further technical developments.

### 3.1 The Mass Function Monad

For our purposes, probabilistic programming languages contain standard control-flow mechanisms and data types, such as our core calculus, together with *probabilistic choice* and *conditioning* operations. In the discrete case, these are given by two effectful operations:

$$\frac{}{\Gamma \vdash_{\text{comp}} \textbf{flip} : \text{bool}} \qquad \frac{\Gamma \vdash t : \mathbb{R}_+}{\Gamma \vdash_{\text{comp}} \textbf{score}\, t : \mathbf{1}}$$

In Bayesian probabilistic programming, we think of **flip** as drawing from a (uniform) prior distribution on bool, and of **score** as recording a likelihood. Typically, one calls **score**$(f(x))$ where $f$ is a density function of a distribution, which records the likelihood of observing data $x$ from the distribution $f$. The score might be zero, a hard constraint: this path is impossible. The score might be in the unit interval, the probability of a discrete observation; but in general a likelihood function can take any positive real value. The inference problem is to approximate the posterior distribution, from the unnormalized posterior defined by the program, combining a prior and likelihood.

To give a set-theoretic semantic structure to such a higher-order language with these two constructs, it suffices to give a monadic interface $\underline{T}$ for which the associativity and unit laws hold, together with two functions:

$$\mathrm{flip} : [\![\mathbf{1}]\!] \to T\,[\![\mathrm{bool}]\!] \qquad\qquad \mathrm{score} : [\![\mathbb{R}_+]\!] \to T\,[\![\mathbf{1}]\!]$$

For the purposes of the discrete development, the following monad fits the bill. A (finite) *mass function* over a set $X$ is a function $\mu : X \to \mathbb{R}_+$ for which there exists a finite set $F \subseteq X$ such that $\mu$ is 0 outside $F$: in other words, the support set $\mathrm{supp}\,\mu \coloneqq \{x \in X \mid \mu(x) \neq 0\}$ is finite. For every set $X$, let $\mathsf{Mass}\,X \coloneqq \{\mu : X \to \mathbb{R}_+ \mid \mu \text{ is a mass function}\}$. The *mass function* monad is given by:

$$\begin{aligned} \underline{\mathsf{Mass}} \coloneqq \textbf{instance}\ \textit{Monad}\,(\mathsf{Mass})\ \textbf{where}& \\ \mathrm{return}\,x_0 &= \lambda x.\ \textbf{if}\ (x = x_0)\ \textbf{then}\ 1\ \textbf{else}\ 0 \\ \mu \gg\!= f \quad &= \lambda y.\ \textstyle\sum_{x \in \mathrm{supp}\,\mu} \mu(x) \cdot (f(x)(y)) \end{aligned}$$

and we set $\mathrm{flip} = \lambda\_.\ \frac{1}{2}$ and $\mathrm{score}\,r = \lambda\{\,() \to r\}$. Intuitively, values of $\mathsf{Mass}\,X$ represent unnormalized probabilistic computations of a result in $X$. From the Bayesian perspective, the meaning of a program is the unnormalized posterior.

Lemma 3.1. *The monadic interface* $\underline{\mathsf{Mass}}$ *defines a ranked monad over* **Set**.

This monad is also known as the *free positive cone monad*, as it constructs the 'positive fragment' of a vector space over the field of reals with basis $X$.

### 3.2 Inference Representations

The mass function semantics is accurate, but idealised: realistic implementations cannot be expected to compute mass functions at arbitrary types, and especially at higher-order types. Instead, probabilistic inference engines would manipulate some representation of the program, while maintaining its semantics.

*Definition 3.2.* A *discrete inference representation* $\underline{T}$ is a sextuple

$$\underline{T} = \left(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}, \mathrm{flip}^{\underline{T}}, \mathrm{score}^{\underline{T}}, m^{\underline{T}}\right)$$

consisting of:

- a monadic interface $\left(T, \mathrm{return}^{\underline{T}}, \gg\!=^{\underline{T}}\right)$;
- two functions $\mathrm{flip}^{\underline{T}} : \mathbb{1} \to T2$ and $\mathrm{score}^{\underline{T}} : \mathbb{R}_+ \to T\mathbb{1}$, where $\mathbb{1} \coloneqq [\![\mathbf{1}]\!]$, $2 \coloneqq [\![\mathrm{bool}]\!]$; and
- an assignment of a *meaning* function $m^{\underline{T}}_X : TX \to \mathsf{Mass}\,X$ for every set $X$

such that the following laws hold for all sets $X$, $Y$, and $x \in X$, $a \in TX$, $r \in \mathbb{R}_+$, and $f : X \to TY$:

$$\mathrm{return}^{\underline{\mathsf{Mass}}}\,x = m(\mathrm{return}^{\underline{T}}\,x) \qquad m(a \gg\!=^{\underline{T}} f) = (m\,a) \gg\!=^{\underline{\mathsf{Mass}}} \lambda x.\ m(f\ x)$$

$$m(\mathrm{flip}^{\underline{T}}) = \mathrm{flip}^{\underline{\mathsf{Mass}}} \qquad m(\mathrm{score}^{\underline{T}}\,r) = \mathrm{score}^{\underline{\mathsf{Mass}}}\,r$$

As with monadic interfaces, we use a type-class notation for defining inference representations.

*Example 3.3 (Discrete weighted sampler).* Consider the type

$$\mathsf{Term}\,\alpha \coloneqq \{\mathsf{Return}\,(\mathbb{R}_+ * \alpha) \mid \mathsf{Flip}\,(\mathsf{Term}\,\alpha * \mathsf{Term}\,\alpha)\}$$

which induces a ranked functor Term. The elements of $\mathsf{Term}\,X$ are binary trees, which we call terms, whose leaves contain weighted values of type $X$. Fig. 6a presents the inference representation structure of the functor Term. Flip represents a probabilistic choice while Return holds the final value and the total weight for the branch. Thus an immediately returning computation is

$$
\begin{array}{l}
\textbf{instance}\ \textit{Discrete Monad}\,(\mathsf{Term})\ \textbf{where}\\
\quad \textbf{return}\,x = \mathsf{Return}\,(1,x)\\
\quad a \gg\!= f\ = \textbf{let}\,(\mathit{scale} : \mathbb{R}_+ * \mathsf{Term}\,X \to \mathsf{Term}\,X) =\ \ \text{-- uses primitive recursion}\\
\qquad\qquad \lambda s.\,\lambda\{\mathsf{Return}\,(r,x) \to \mathsf{Return}\,(s\cdot r, x)\\
\qquad\qquad\quad \big|\mathsf{Flip}\,(k_{\mathsf{False}},k_{\mathsf{True}}) \to \mathsf{Flip}\,(\mathit{scale}(s,k_{\mathsf{False}}),\mathit{scale}(s,k_{\mathsf{True}}))\}\\
\qquad\quad \textbf{in}\ \ \textbf{match}\,a\,\textbf{with}\,\{\\
\qquad\qquad\quad \mathsf{Return}\,(r,x) \to \mathit{scale}(r, f\ x)\\
\qquad\qquad\quad \big|\mathsf{Flip}\,(k_{\mathsf{False}},k_{\mathsf{True}}) \to \mathsf{Flip}\ (k_{\mathsf{False}} \gg\!= f,\ \text{-- uses primitive recursion}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k_{\mathsf{True}} \gg\!= f)\}\\
\quad \textbf{flip}\ \ = \mathsf{Flip}\,(\mathsf{Return}\,(1,\mathsf{False}),\mathsf{Return}\,(1,\mathsf{True}))\\
\quad \textbf{score}\,r\ = \mathsf{Return}\,(r,())\\
\quad m\,a\ \ \ = \textbf{fold}\,\lambda\{\mathsf{Return}\,(r,x) \to \mathsf{Mass}\,.\textbf{do}\,\{\textbf{score}\,r;\textbf{return}\,x\}\\
\qquad\qquad\quad \big|\mathsf{Flip}\,(\mu_{\mathsf{False}},\mu_{\mathsf{True}}) \to \mathsf{Mass}\,.\textbf{do}\,\{x \leftarrow \textbf{flip};\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ x\ \textbf{then}\ \mu_{\mathsf{True}}\ \textbf{else}\ \mu_{\mathsf{False}}\}\}
\end{array}
$$

(a) Discrete weighted sampler representation

$$
\begin{array}{l}
\textbf{instance}\ \textit{Discrete Monad}\,(\mathsf{Enum})\ \textbf{where}\\
\quad \textbf{return}\,x = [(1,x)]\\
\quad x_s \gg\!= f = \textbf{let}\,(\mathit{scale} : \mathbb{R}_+ * \mathsf{Enum}\,X \to \mathsf{Enum}\,X) =\\
\qquad\qquad \lambda\{(r,x_s) \to \mathrm{map}\,\lambda\{(s,y) \to (r\cdot s, y)\}\\
\qquad\qquad\qquad\qquad x_s\}\\
\qquad\quad \textbf{in}\ \ \mathrm{foldr}\,[\,]\\
\qquad\qquad \lambda\{((r,x),y_s) \to \mathit{scale}(r, f\ x) +\!\!+\ y_s\}\\
\qquad\qquad x_s\\
\quad \mathrm{flip}\ \ = [(\tfrac{1}{2},\mathsf{False}),(\tfrac{1}{2},\mathsf{True})]\\
\quad \mathrm{score}\,r\ = [(r,())]\\
\quad m\,x_s\ \ \ = \lambda a.\ \ \ \text{-- } m\,x_s\ a = \sum_{\substack{(r,x)\in x_s\\ x=a}} r\\
\qquad\qquad \mathrm{foldr}\,0\left(\begin{array}{l}\lambda\{((r,x),s) \to\\ \quad \textbf{if}\ x = a\ \textbf{then}\ r+s\ \textbf{else}\ s\}\end{array}\right) x_s
\end{array}
$$

(b) Discrete enumeration sampler

$$
\begin{array}{l}
\textbf{instance}\ \textit{Inf Trans}\,(\mathsf{W})\ \textbf{where}\\
\quad \textbf{lift}_{\underline{T}}\,a\ \ \ \ \ \ = \underline{T}.\textbf{do}\,\{x \leftarrow a;\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}(1,x)\}\\
\quad \textbf{return}_{\mathsf{W}\,\underline{T}}\,x = \mathrm{return}^{\underline{T}}(1,x)\\
\quad a \gg\!=_{\mathsf{W}\,\underline{T}} f\ = \underline{T}.\textbf{do}\,\{(r,x) \leftarrow a;\\
\qquad\qquad\qquad\qquad\qquad\qquad (s,y) \leftarrow f(x);\\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}(r\cdot s, y)\}\\
\quad \textbf{flip}_{\mathsf{W}\,\underline{T}}\ \ \ \ \ = \textbf{lift}\,\textbf{flip}^{\underline{T}}\\
\quad \textbf{score}_{\mathsf{W}\,\underline{T}}\,r\ = \mathrm{return}^{\underline{T}}(r,())\\
\quad m_{\mathsf{W}\,\underline{T}}\,a\ \ \ \ = \lambda x.\ \textstyle\sum_{(r,x)\in \mathrm{supp}\,m^T(a)} r\\
\quad (\textbf{tmap}\,\underline{t})_X\ \ = \underline{t}_{\mathbb{R}_+ * X}
\end{array}
$$

(c) Discrete weighting transformer

Fig. 6. Example inference representations (a,b) and transformers (c)

represented by a leaf with weight 1. The auxiliary function *scale* in the definition of $\gg\!=$ scales the leaves of its input term by the input weight. The function $\gg\!=$ itself substitutes terms for the leaves according to its input function $f$, making sure the newly grafted terms are scaled appropriately. The probabilistic choice operation **flip** constructs a single node with each leaf recording the probabilistic choice *unweighted*. Conditioning records the input weight.

The meaning function recurses over the term, replacing each node representing a probabilistic choice by probabilistic choice of the mass function monad, and reweighting the end result appropriately.

The main step in validating the inference representation laws involves $\gg\!=$: first show that composing the meaning function with the auxiliary function *scale* scales the meaning of the input term appropriately, and then proceed by structural induction on terms.

The weighted sampler representation in fact forms a proper monad over **Set**: it is the free monad for an algebraic theory with a binary operation flip and unary operations $\mathrm{score}_r$ subject to $\mathbf{flip}(\mathbf{score}_r(x), \mathbf{score}_r(y)) = \mathbf{score}_r(\mathbf{flip}(x, y))$. As the mass function monad also validates these equations, the meaning function is then the unique monad morphism from Term to Mass preserving the operations **flip** and **score**.

However, we emphasise that an inference representation need not form a proper monad, and that the meaning function need not be a monad morphism. Indeed, the Pop Sam representation introduced in Sec. 6 is not a monad and most of the non-trivial inference transformations we discuss are not monad morphisms.

The weighted sampler representation allows us to incorporate both intensional and operational aspects into our development. Bayesian inference ultimately reduces a representation into probabilistic simulation. The weighted sampler representation can thus act as an internal representation of this simulation. Moreover, its continuous analogue will allow us to manipulate traces when analysing the Trace Markov Chain Monte Carlo algorithm in Sec. 7.

*Example 3.4 (Enumeration).* The type $\mathsf{Enum}\,\alpha \coloneqq \mathsf{List}(\mathbb{R}_+ * \alpha)$ induces a ranked functor Enum. Elements of $\mathsf{Enum}\,X$ form an enumeration of the mass function they represent, with the same value $x$ potentially appearing multiple times with different weights. Values not appearing in the list at all have weight 0.

Fig. 6b presents an inference representation structure using Enum. Returning a value lists the unique non-zero point mass. The $\gg\!=$ operation applies the given function to each element listed, scales the list appropriately and accumulates all intermediate lists. The choice operation enumerates both branches with equal probability, and conditioning inserts a scaling factor. The meaning function assigns to an element the sum of its weights. This definition uses an equality predicate.

Establishing the inference representation laws is straightforward.

### 3.3 Inference Transformations

We can now define the central validity criterion in our development. We decompose Bayesian inference algorithms into smaller transformations between inference representations. To be correct, these transformations need to preserve the meaning of the representation they manipulate:

*Definition 3.5.* Let $\underline{T}$, $\underline{S}$ be two inference representations. A *discrete inference transformation* $\underline{t} : \underline{T} \to \underline{S}$ assigns to each set $X$ a function $\underline{t}_X : TX \to SX$ satisfying $m^{\underline{T}}(a) = m^{\underline{S}}(\underline{t}_X(a))$ for every $a \in \underline{T}X$.

This validity criterion guarantees nothing beyond the preservation of the overall mass function of our representation. The transformed representation may not be better for inference along any axis, such as better convergence properties or execution time. It is up to the inference algorithm designer to convince herself of such properties by other means: formal, empirical, or heuristic.

Some transformations change the representation type:

*Example 3.6 (Enumeration).* Define a transformation: $\underline{t} : \underline{\mathsf{Term}} \to \underline{\mathsf{Enum}}$ by:

$$\begin{aligned}\underline{t} \coloneqq \lambda\{\ &\mathsf{Return}\,(r, x) && \to \mathsf{Enum}\,.\mathbf{do}\,\{\mathbf{score}\,r; \mathbf{return}\,x\}\\ \mid\ &\mathsf{Flip}\,(x_s^{\mathsf{False}}, x_s^{\mathsf{True}}) && \to \mathsf{Enum}\,.\mathbf{do}\,\{b \leftarrow \mathbf{flip}; \mathbf{if}\ b\ \mathbf{then}\ x_s^{\mathsf{True}}\ \mathbf{else}\ x_s^{\mathsf{False}}\}\}\end{aligned}$$

Straightforward calculation shows it preserves the meaning functions.

The last example is a special case: analogous functions form inference transformations $\underline{t_{\underline{T}}} : \underline{\mathsf{Term}} \to \underline{T}$ for every discrete inference representation $\underline{T}$. To establish meaning preservation, calculate that both $m^{\underline{\mathsf{Term}}}$ and $m^{\underline{T}} \circ \underline{t_{\underline{T}}}$ are monad morphisms that preserve probabilistic choice and conditioning and appeal to the initiality of $\underline{\mathsf{Term}}$.

An inference transformation need not be natural:

*Example 3.7 (Aggregation).* Recall the functions $\mathrm{aggr}_X : \mathsf{List}(\mathbb{R}_+ * X) \to \mathsf{List}(\mathbb{R}_+ * X)$ from Ex. 2.5 which aggregate list elements according to their $X$ component by summing their weights. It forms an inference transformation $\mathrm{aggr} : \underline{\mathsf{Enum}} \to \underline{\mathsf{Enum}}$. The meaning preservation proof uses straightforward structural induction. Note that aggr is not a natural transformation.

### 3.4 Inference Transformers

We can decompose the weighted sampler representation $\underline{\mathsf{Term}}$, which forms a monad, by transforming the *discrete sampler* representation $\mathsf{DSam}\,X := \{\mathsf{Return}\,X \mid \mathsf{Sample}\,(\mathsf{DSam}\,X * \mathsf{DSam}\,X)\}$ with the following *writer monad transformer* $\mathsf{W}\,T\,X := T(\mathbb{R}_+ * X)$, i.e. $\mathsf{Term} = \mathsf{W}\,\mathsf{DSam}$. Such decompositions form basic building blocks for constructing and reasoning about more sophisticated representations.

*Definition 3.8.* An *inference transformer* $\underline{F}$ is a triple $(F, \mathrm{tmap}^{\underline{F}}, \mathrm{lift}^{\underline{F}})$ whose components assign:

- inference representation $F\,\underline{T}$ to every inference representation $\underline{T}$;
- inference transformation $\mathrm{tmap}^{\underline{F}}\,\underline{t} : \underline{FT} \to \underline{FS}$ to every inference transformation $\underline{t} : \underline{T} \to \underline{S}$; and
- inference transformation $\mathrm{lift}_T : \underline{T} \to F\,\underline{T}$ to every inference representation $\underline{T}$.

We use type-class notation for defining inference transformers.

*Example 3.9.* The *weighting* inference transformer structure on $\mathsf{W}\,T\,X := T(\mathbb{R}_+ * X)$ is given in Fig. 6c. We lift a representation in $\underline{T}$ into $\mathsf{W}\,\underline{T}$ by assigning weight 1 to it. The monadic interface uses the standard writer monad for the multiplication structure on $\mathbb{R}_+$, accumulating the weights as computation proceeds. We lift the probabilistic choice from $\underline{T}$, but crucially we reimplement a *new* conditioning operation using the explicitly given weights. The mass function meaning of a representation then accumulates the mass of all weights associated to a given value. We transform an inference transformation by picking the component of the appropriate type.

It is straightforward to show that $\mathsf{W}\,\underline{T}$ is an inference representation, using preservation of return and $\gg=$ by the meaning function to reduce the proof to manipulations of weighted sums over $\mathbb{R}_+$. Establishing the validity of lift and tmap is straightforward.

The weighting transformer *augments* the representation with a new conditioning operation, but *transforms* its choice operation to the new representation. We will later see more examples of both kinds.

### 3.5 Summary

We have introduced our three core abstractions, inference representations, transformations, and transformers, in relation to a mathematical semantic structure, the mass function monad. The examples so far show that the higher-order structure in our core calculus acts as a useful glue for manipulating and defining these abstractions. In the continuous case, we will also use this higher-order structure to represent computations over the real numbers.

## 4 PRELIMINARIES

In order to generalise this higher-order treatment to the continuous case, we first need to review and develop the mathematical theory of quasi-Borel spaces. Our development uses Kock's synthetic measure theory [2012], which allows us to reason analogously to measure theory. In order to present the synthetic theory, we briefly review the required category theoretic concepts. These sections are aimed at readers who are interested in the categorical context of our development. Other readers may continue directly to § 4.3.

### 4.1 Category Theory

*Basic Notions.* We assume basic familiarity with categories $\mathcal{C}$, $\mathcal{D}$, functors $F, G : \mathcal{C} \to \mathcal{D}$, and natural transformations $\alpha, \beta : F \to G$, and their theory of limits, colimits, and adjunctions. To fix notation, a *cartesian closed category* is a category with finite products, denoted by $\mathbb{1}$, $\times$, $\prod_{i=1}^{n}$, and exponentials, denoted by $X^Y$. In this subsection, we use the fragment of our core calculus consisting of the simply-typed $\lambda$-calculus (with sums, if necessary) to more compactly review the relevant concepts.

*Monads.* A *strong monad* $\underline{T}$ over a cartesian closed category is a triple $(T, \text{return}, \gg\!=)$ consisting of an assignment of an object $TX$ and a morphism $\text{return}_X : X \to TX$ for every object $X$, and an assignment of a morphism $\gg\!=_{X,Y}\colon TX \times (TY)^X \to TY$, satisfying the monad laws from § 2.5. Given a monad $\underline{T}$, a *$\underline{T}$-algebra* $A$ is a pair $(|A|, \gg\!=^A)$ consisting of an object $|A|$, called the *carrier*, and an assignment of a morphism $\gg\!=^A_X\colon |A|^X \to |A|^{TX}$ to every object $X$ satisfying

$$(\text{return}\, x \gg\!=^A f) = f\ x \quad \text{and} \quad ((a \gg\!= f) \gg\!=^A g) = a \gg\!= (\lambda x.\, f(x) \gg\!=^A g).$$

The pair $(T\,X, \gg\!=)$ always forms a $T$-algebra called the *free* $\underline{T}$-algebra over $X$. The *Eilenberg-Moore* category $\mathcal{C}^{\underline{T}}$ for a monad $\underline{T}$ consists of $\underline{T}$-algebras and their homomorphism. The *Kleisli* category $\mathcal{C}_{\underline{T}}$ consists of the same objects as $\mathcal{C}$, but morphisms from $X$ to $Y$ in $\mathcal{C}_{\underline{T}}$ are morphisms $X \to T\,Y$ in $\mathcal{C}$. The Kleisli category $\mathcal{C}_{\underline{T}}$ inherits any coproducts $\mathcal{C}$ has. A strong monad $\underline{T}$ is *commutative* when, for every

$$a : \underline{T}X, b : \underline{T}Y \vdash \quad \underline{T}.\mathbf{do}\,\{x \leftarrow a; y \leftarrow b; \mathbf{return}(x, y)\} \quad = \quad \underline{T}.\mathbf{do}\,\{y \leftarrow b; x \leftarrow a; \mathbf{return}(x, y)\}$$

(The notion of strong/commutative monad is due to [Kock 1972]; our formulation of algebras also appears in [Marmolejo and Wood 2010].)

*Biproducts.* A zero object $\mathbb{Z}$ is both initial and terminal. A category has *(finite, countable, etc.) biproducts* if it has a zero object (and hence zero morphisms $\mathbf{0}_{X,Y} : X \to \mathbb{Z} \to Y$) and the following canonical morphisms are invertible:

$$\left[\left(\delta_{i,j}\right)_{j\in I}\right]_{i\in I} : \textstyle\sum_{i\in I} X_i \to \prod_{j\in I} X_j \qquad \text{where: } \delta_{i,i} \coloneqq \text{id}_{X_i},\ \delta_{i,j} \coloneqq \mathbf{0}_{X_i,X_j} \text{ for } i \neq j.$$

*Algebraic Structure.* Recall the notion of a *commutative monoid* $(M, 1, \cdot)$ in a category with finite products. We extend it to countably many arguments. Let $\mathcal{C}$ be a category with countable products. A *$\sigma$-monoid* (see also [Haghverdi and Scott 2006]) is a triple $(M, 0, \Sigma)$ consisting of: an object $M$; a morphism $0 : \mathbb{1} \to M$; and a morphism $\Sigma : M^{\mathbb{N}} \to M$ such that:

- setting $\delta_0 \coloneqq \text{id}_M : M \to M$ and $\delta_i \coloneqq 0 \circ !\, : M \to \mathbb{1} \to M$, $i > 0$, we have $\Sigma \circ (\delta_i)_{i\in\mathbb{N}} = \delta_0$; and
- for every bijection $\varphi : \mathbb{N} \cong \mathbb{N} \times \mathbb{N}$, $a_{(-,-)} : M^{\mathbb{N}\times\mathbb{N}} \vdash \Sigma\left(\Sigma\left(a_{(i,j)}\right)_{j\in\mathbb{N}}\right)_{i\in\mathbb{N}} = \Sigma\left(a_{\varphi(k)}\right)_{k\in\mathbb{N}}$.

Proposition 4.1. *In a category with countable biproducts, each object $M$ is a $\sigma$-monoid via:*

$$\mathbf{0}_{\mathbb{1},M} : \mathbb{1} \to M \qquad\qquad \Sigma : \textstyle\prod_{i\in\mathbb{N}} M \cong \sum_{i\in\mathbb{N}} M \xrightarrow{\nabla} M \text{ where } \nabla \text{ is the codiagonal.}$$

*Every morphism is a $\sigma$-monoid homomorphism with respect to this structure.*

A *$\sigma$-semiring* is a quintuple $(S, 1, \cdot, 0, \Sigma)$ consisting of: a commutative monoid $(S, 1, \cdot)$; and a $\sigma$-monoid $(S, 0, \Sigma)$, such that $a : S, b\_ \in S^{\mathbb{N}} \vdash a \cdot \Sigma\,(b_i)_{i\in\mathbb{N}} = \Sigma\,(a \cdot b_i)_{i\in\mathbb{N}}$. Given a $\sigma$-semiring $(S, 1, \cdot, 0, \Sigma)$, an *$S$-module* is a pair $(M, \odot)$ consisting of a $\sigma$-monoid $M$; and a morphism $\odot : S \times M \to M$ satisfying: $x = x, 0_S \odot x = 0_M, (a \cdot b) \odot x = a \odot (b \odot x), \left(\Sigma\,(a_n)_{n\in\mathbb{N}}\right) \odot x = \Sigma\,(a_n \odot x)_{n\in\mathbb{N}}$.

### 4.2 Synthetic Measure Theory

Synthetic mathematics identifies structure and axioms from which we can recover the main concepts and results of specific mathematical theories, and transport them to new settings. We now briefly recount the relevant parts of Kock's [2012] development. (In the finite discrete case, this is also related to Jacobs's [2017] work on effectuses.)

*4.2.1 Axioms and Structure.* Let $C$ be a cartesian closed category with countable products and coproducts, and let $\underline{\mathrm{M}}$ be a commutative monad over $C$. If the morphism $! : \mathrm{M}\,\mathbb{0} \to \mathbb{1}$ is invertible, then both the Eilenberg-Moore category $C^{\underline{\mathrm{M}}}$ and the Kleisli category $C_{\underline{\mathrm{M}}}$ have zero objects. As a consequence, we have a canonical $\underline{\mathrm{M}}$-homomorphism $\gg\!= \left[\left(\delta_{i,j}\right)_j\right]_i : \mathrm{M} \sum_{i\in\mathbb{N}} X_i \to \prod_{j\in\mathbb{N}} \mathrm{M}\, X_j$.

*Definition 4.2.* A *measure category* is a pair $(C, \underline{\mathrm{M}})$ consisting of a cartesian closed category $C$ with countable products and coproducts, and equalisers; and a commutative monad $\underline{\mathrm{M}}$ over $C$ such that the morphisms $! : \mathrm{M}\,\mathbb{0} \to \mathbb{1}$ and $\gg\!= \left[\left(\delta_{i,j}\right)_j\right]_i : \mathrm{M} \sum_{i\in\mathbb{N}} X_i \to \prod_{j\in\mathbb{N}} \mathrm{M}\, X_j$ are invertible.

We fix a measure category $(C, \underline{\mathrm{M}})$ for the remainder of this section. The intuition is that $\mathrm{M}\,X$ is the object of distributions/measures over $X$. Kock shows that, while short, the above definition has surprisingly many consequences.

Both the Eilenberg-Moore and the Kleisli categories have countable biproducts, and as a consequence, all $\underline{\mathrm{M}}$-algebras have a $\sigma$-monoid structure and all $\underline{\mathrm{M}}$-homomorphisms are $\sigma$-monoid homomorphisms with respect to it. Moreover, this structure on the free algebra on the terminal object $R \coloneqq \mathrm{M}\,\mathbb{1}$ extends to a $\sigma$-semiring structure by setting: $1 \coloneqq \mathbf{return}()$ and $r \cdot s \coloneqq \underline{\mathrm{M}}.\mathbf{do}\,\{r; s\}$. Kock calls this structure the *$\sigma$-semiring of scalars*. Each $\underline{\mathrm{M}}$-algebra $A$ has an $R$-module structure:

$$r : R, a : |A| \vdash r \odot a \coloneqq \underline{\mathrm{M}}.\mathbf{do}\,\{r; a\}$$

As $C$ has equalisers, for each object $X$, we may form the equaliser $\mathrm{P}\,X \overset{\mathrm{sub}_X}{\dashrightarrow} \mathrm{M}\,X \underset{\underline{1}}{\overset{\mathrm{M!}}{\rightrightarrows}} R$ because $R = \mathrm{M}\,\mathbb{1}$. Each $\mathrm{sub}_X$ is monic, the monadic structure factors through sub turning P into a commutative monad $\underline{\mathrm{P}}$, and $\mathrm{sub} : \underline{\mathrm{P}} \rightarrowtail \underline{\mathrm{M}}$ into a strong monad monomorphism.

The morphism $\mathrm{M!} : \mathrm{M}\,X \to R$ is called the *total measure* morphism, and P is then the sub-object of all the measures with total measure 1, and so we think of it as the object of *probability measures* over $X$. For example, every $\underline{\mathrm{P}}$-algebra is closed under *convex* linear combinations of scalars: if $r_- : \mathbb{N} \to R$ satisfies $\Sigma\,(r_i)_i = 1$ then $\underline{\mu}_{r_-} : (\mathrm{P}\,X)^{\mathbb{N}} \vdash \mathrm{M!}(\Sigma\left(r_i \odot \underline{\mu}_i\right)_i) = 1$.

*4.2.2 Notation and Basic Properties.* Kock's theory shines brightly when we adopt a measure-theoretic notation, as in Fig. 7, by thinking of $\mathrm{M}\,X$ as the object of measures over $X$, and $R$ as the object of scalars these measures take values in. The functorial action of the monad allows us to push measures along morphisms, and pushing all the measure into the terminal object gives a scalar we think of as the total measure of an object. The monadic return acts as a dirac distribution. The main advantage is the *Kock integral*, synonymous to the monadic $\gg\!=$. The main difference between the Kock integral $\oint$ and the usual Lebesgue integral $\int$ from measure theory is that the Kock integral evaluates to a *measure*, and not a scalar. Calculating with the Kock integral is analogous to using Lebesgue integrals with respect to a generic test function, and proceeding by algebraic manipulation. The scalar rescaling $\odot$ allows us to rescale a distribution by an arbitrary weight function. A *kernel* is a morphism $k : X \to \mathrm{M}\,Y$, and we use the usual notation for integration against a kernel and iterated integration. We define the product measure by iterated integration. Finally, the $\gg\!=$ operation of an $\underline{\mathrm{M}}$-algebra $A$ gives rise to an expectation operation. Here we will only make use of the scalars' algebra structure, which generalises the usual Lebesgue integral.

| | Notation | Meaning | Terminology |
|---|---|---|---|
| | $R$ | $\coloneqq \mathrm{M}\,\mathbb{1}$ | Scalars |
| $f : Y^X, \underline{\mu} : \mathrm{M}\,X$ | $\vdash f_*\underline{\mu}$ | $\coloneqq (\mathrm{M}\,f)(\underline{\mu})$ | Push-forward |
| $\underline{\mu} : \mathrm{M}\,X$ | $\vdash \underline{\mu}(X)$ | $\coloneqq !_*\underline{\mu}$ | The total measure |
| $x : X$ | $\vdash \underline{\delta}_x$ | $\coloneqq \mathbf{return}(x)$ | Dirac distribution |
| $\underline{\mu} : \mathrm{M}\,X, f : (\mathrm{M}\,Y)^X$ | $\vdash \int_X f(x)\underline{\mu}(\mathrm{d}x)$ | $\coloneqq \underline{\mu} \gg\!= f$ | Kock integral |
| $w : R^X, \underline{\mu} : \mathrm{M}\,X$ | $\vdash w \odot \underline{\mu}$ | $\coloneqq \int_X (w(x) \odot \underline{\delta}_x)\underline{\mu}(\mathrm{d}x)$ | Rescaling |
| $\left[\begin{array}{l} f : (TZ)^{X\times Y}, \\ x : X, k : (TY)^X \end{array}\right]$ | $\vdash \int_Y f(x,y)k(x,\mathrm{d}y)$ | $\coloneqq \int_Y f(x,y)k(x)(\mathrm{d}y)$ | Kernel integration |
| $\left[\begin{array}{l} f : (\mathrm{M}\,X)^{X\times Y}, \\ \underline{\mu} \in \mathrm{M}(X\times Y) \end{array}\right]$ | $\vdash \iint_{X\times Y} f(x,y)\underline{\mu}(\mathrm{d}x,\mathrm{d}y)$ | $\coloneqq \int_{X\times Y} f(z)\underline{\mu}(\mathrm{d}z)$ | Iterated integrals |
| $\underline{\mu} : \mathrm{M}\,X, \underline{\nu} : \mathrm{M}\,Y$ | $\vdash \underline{\mu} \otimes \underline{\nu}$ | $\coloneqq \int_X \left(\int_Y \underline{\delta}_{(x,y)}\underline{\nu}(\mathrm{d}y)\right)\underline{\mu}(\mathrm{d}x)$ | Product measure |
| $\underline{\mu} : \mathrm{M}\,X, f : \lvert A\rvert^X$ | $\vdash \mathbb{E}^A_{x\sim\underline{\mu}}[f(x)]$ | $\coloneqq \underline{\mu} \gg\!= f$ | Expectation |
| $f : R^X, \underline{\mu} : \mathrm{M}\,X$ | $\vdash \int_X f(x)\underline{\mu}(\mathrm{d}x)$ | $\coloneqq \mathbb{E}^R_{x\sim\underline{\mu}}[f(x)]$ | Lebesgue integral |

Fig. 7. Synthetic measure theory notation

The justification for this notation is that it obeys the expected properties, which we now survey. The commutativity of the monad lets us change the order of integration:

Theorem 4.3 (Fubini-Tonelli). *For every pair of objects $X$, $Y$ in a measure category $(C, \underline{\mathrm{M}})$:*

$$\iint_{X\times Y} f(x,y)(\underline{\mu} \otimes \underline{\nu})(\mathrm{d}x,\mathrm{d}y) \quad = \quad \int_Y \underline{\nu}(\mathrm{d}y) \int_X \underline{\mu}(\mathrm{d}x) f(x,y) \quad = \quad \int_Y \underline{\nu}(\mathrm{d}y) \int_X \underline{\mu}(\mathrm{d}x) f(x,y)$$

*Moreover, for every* M*-algebra $A$:*

$$\underline{\mu} : \mathrm{M}\,X, \underline{\nu} : \mathrm{M}\,Y, f : \lvert A\rvert^{X\times Y} \vdash \mathop{\mathbb{E}^A}_{\substack{x\sim\underline{\mu}\\ y\sim\underline{\nu}}}[f(x,y)] = \mathop{\mathbb{E}^A}_{x\sim\underline{\mu}}[\mathop{\mathbb{E}^A}_{y\sim\underline{\nu}}[f(x,y)]] = \mathop{\mathbb{E}^A}_{y\sim\underline{\nu}}[\mathop{\mathbb{E}^A}_{x\sim\underline{\mu}}[f(x,y)]]$$

As usual, we allow placing the binder $\underline{\mu}(\mathrm{d}x)$ on either side of the integrand $f(x)$.

Integrals and expectation interact well with the $R$-module structure in the sense that they are homomorphisms in both arguments. The precise statement of this fact can be found in Appendix A.

The push-forward operation interacts with rescaling in the following way:

Theorem 4.4 (Frobenius reciprocity). *For all objects $X$, $Y$ in a measure category $(C, \underline{\mathrm{M}})$:*

$$w : R^X, \underline{\mu} : \mathrm{M}\,X, f : Y^X \vdash w \odot \left(f_*\underline{\mu}\right) = f_*\left((w \circ f) \odot \underline{\mu}\right)$$

When calculating in this notation, we use the equations in Appendix A where we present a toolbox for synthetic measure theory. This toolbox includes most of the equations we come to expect from standard measure theory, like the change of variables law. To validate them, inline the definitions and proceed using the usual category-theoretic properties.

The following two sections contain relevant extensions to Kock's theory.

*4.2.3 Radon-Nikodym Derivatives.* The Radon-Nikodym Theorem is a powerful tool in measure theory, and we now phrase a synthetic counterpart. As usual in the synthetic setting, we set the definitions up such that the theorem will be true. Doing so highlights the difference between three measure-theoretic concepts that coincide in measure theory, but may differ in the synthetic setting.

Let $\underline{\mu}, \underline{\nu} \in \mathrm{M}\,X$ be measures. We say that $\underline{\nu}$ is *absolutely continuous* with respect to $\underline{\mu}$, and write $\underline{\nu} \ll \underline{\mu}$, when there exists a morphism $w : X \to R$ such that $\underline{\nu} = w \odot \underline{\mu}$. Given two morphisms $w, v : X \to R$ and a measure $\underline{\mu} \in \mathrm{M}\,X$, we say that $w$ and $v$ are *equal $\underline{\mu}$-almost everywhere* ($\underline{\mu}$-a.e.) when $w \odot \underline{\mu} = v \odot \underline{\mu}$. A *measurable property* over $X$ is a morphism $P : X \to \mathrm{bool}$. Given a measure $\underline{\mu} \in \mathrm{M}\,X$ a measurable property $P$ over $X$ *holds $\underline{\mu}$-a.e.*, when the morphism $[P] \coloneqq \lambda x.\ \mathbf{if}\ P\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0$ is equal $\underline{\mu}$-a.e. to 1.

Theorem 4.5 (Radon-Nikodym). *Let $(C, \mathrm{M})$ be a well-pointed measure category. For every $\underline{\nu} \ll \underline{\mu}$ in $\mathrm{M}\,X$, there exists a $\underline{\mu}$-a.e. unique morphism $\frac{\mathrm{d}\underline{\nu}}{\mathrm{d}\underline{\mu}} : X \to R$ satisfying $\frac{\mathrm{d}\underline{\nu}}{\mathrm{d}\underline{\mu}} \odot \underline{\mu} = \underline{\nu}$.*

*4.2.4 Kernels.* We say that a kernel $k : X \to \mathrm{M}\,Y$ is *Markov* when, for all $x$, $k(x, Y) = 1$, i.e., when $k$ factors through the object of probability measures via $\mathrm{sub} : \mathrm{P} \rightarrowtail \mathrm{M}$. We now restrict attention to kernels $k : X \to \mathrm{M}\,X$ over the same object $X$. We say that such a kernel *preserves* a measure $\underline{\mu}$ when $\underline{\mu} \gg\!= k = \underline{\mu}$. Recall the morphism $\mathrm{swap} \coloneqq \lambda(x, y).\ (y, x) : X \times Y \to Y \times X$. Given a measure $\underline{\mu} \in \mathrm{M}\,X$ and a kernel $k$, we define the *box product* by $\underline{\mu} \boxtimes k \coloneqq \iint_{X \times X} \underline{\delta}_{(x,y)} \underline{\mu}(\mathrm{d}x) k(x, \mathrm{d}y)$. A kernel $k$ is *reversible* with respect to a measure $\underline{\mu} \in \mathrm{M}\,X$ when $\mathrm{swap}_*(\underline{\mu} \boxtimes k) = \underline{\mu} \boxtimes k$.

The following standard results on kernels transfer into the synthetic setting. If a *Markov* kernel $k$ is reversible with respect to $\underline{\mu}$, then $k$ preserves $\underline{\mu}$. Kernels obtained by rescaling the Dirac kernel, i.e., $\lambda x.\ w(x) \odot \underline{\delta}_x$ are reversible w.r.t. all measures. Finally, linear combinations $\lambda x.\ \sum_{n \in \mathbb{N}} r_n \odot k_n(x)$ of reversible kernels w.r.t. $\underline{\mu}$ are also reversible w.r.t. $\underline{\mu}$.

### 4.3 Quasi-Borel Spaces

It remains to show that there is a concrete model of synthetic measure theory that contains the classical measure theoretic ideas that are central to probability theory and inference. This is novel because Kock's work [2012] is targeted at the geometric/topological setting, whereas probability theory is based around Borel sets rather than open sets. It is non-trivial because the traditional setting for measure theory does not support higher-order functions [Aumann 1961] and commutativity of integration is subtle in general. In this section we resolve these problems by combining some recent discoveries [Heunen et al. 2017; Staton 2017], and exhibit a model of synthetic measure theory which contains classical measure theory, for instance:

- the $\sigma$-semiring over the morphisms $\mathbb{1} \to R$ is isomorphic to the usual $\sigma$-semiring over the extended non-negative reals, $\overline{\mathbb{R}}_+$;
- this isomorphism induces a bijective correspondence between the morphisms $R \to \mathbb{1} + \mathbb{1}$ and the Borel subsets of $\overline{\mathbb{R}}_+$, as characteristic functions, and also between the morphisms $R \to R$ and the measurable functions $\overline{\mathbb{R}}_+ \to \overline{\mathbb{R}}_+$;
- it also induces an injection of the morphisms $\mathbb{1} \to \mathrm{M}(R)$ into the set of Borel measures on $\overline{\mathbb{R}}_+$, whose image contains all the probability measures; the morphisms $R \to \mathrm{M}(R)$ include all the Borel probability kernels;
- the canonical morphism $R^R \times \mathrm{M}(R) \to R$, $(f, \underline{\mu}) \mapsto \int f(x)\, \underline{\mu}(\mathrm{d}x)$, corresponds to classical Lebesgue integration.

Moreover, each object $X$ can be seen as a set $U(X) = C(\mathbb{1}, X)$ with structure, because the category is well-pointed, in the sense that the morphisms $X \to Y$ are a subset of the functions $U(X) \to U(Y)$.

*4.3.1 Rudiments of Classical Measure Theory.* Measurable spaces are the cornerstone of conventional measure theory, supporting a notion of measure.

Recall that a *$\sigma$-algebra* on a set $X$ is a set $\Sigma_X$ of subsets of $X$ that is closed under countable unions and complements. A *measurable space* is a set together with a $\sigma$-algebra. A *measure* is a $\sigma$-additive

function $\Sigma_X \to \overline{\mathbb{R}}_+$. A function $f$ between measurable spaces is *measurable* if the inverse image of every measurable set according to $f$ is measurable.

For example, on a Euclidean space $\overline{\mathbb{R}}^n$ we can consider the Borel sets, which form the smallest $\sigma$-algebra containing the open cubes. There is a canonical measure on $\overline{\mathbb{R}}^n$, the *Lebesgue measure*, which assigns to each cube its volume, and thus to every measurable function $f\colon \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}_+$ a Lebesgue integral $\int_{\overline{\mathbb{R}}^n} f \in \overline{\mathbb{R}}_+$. A slightly more general class of measures is the *$\sigma$-finite measures*, which include the Lebesgue measures and are closed under disjoint unions and product measures.

A measurable space that arises from the Borel sets of a Polish space is called a *standard Borel space*. In fact, every standard Borel space is either countable or isomorphic to $\overline{\mathbb{R}}$. Standard Borel spaces are closed under countable products and countable disjoint unions.

*4.3.2 Quasi-Borel Spaces.* In this section we fix an uncountable standard Borel space, $\mathfrak{R}$. For example, $\mathfrak{R} = \overline{\mathbb{R}}$. The basic idea of quasi-Borel spaces is that rather than focusing on measurable sets of a set $X$, as in classical measure theory, one should focus on the admissible random elements $\mathfrak{R} \to X$.

*Definition 4.6 ([Heunen et al. 2017]).* A *quasi-Borel space (QBS)* is a set $X$ together with a set of functions $M_X \subseteq [\mathfrak{R}, X]$ such that $(i)$ all the constant functions are in $M_X$, $(ii)$ $M_X$ is closed under precomposition with measurable functions on $\mathfrak{R}$, and $(iii)$ $M_X$ satisfies the piecewise condition: if $\mathfrak{R} = \biguplus_{i=1}^{\infty} U_i$, where $U_i$ is Borel measurable and $\alpha_i \in M_X$ for all $i$, then $\biguplus_{i=1}^{\infty} \alpha_i \cap (U_i \times X)$ is in $M_X$.

A *morphism* $f : X \to Y$ is a function that respects the structure, i.e. if $\alpha \in M_X$ then $(f \circ \alpha) \in M_Y$. Morphisms compose as functions, and we have a category **QBS**.

A QBS $X$ is a *subspace* of a QBS $Y$ if $X \subseteq Y$ and $M_X = \{\alpha : \mathfrak{R} \to X \mid \alpha \in M_Y\}$.

A measurable space $X$ can be turned into a QBS when given the set of measurable functions $\mathfrak{R} \to X$ as $M_X$. When $X$ and $Y$ are standard Borel spaces considered as QBSes this way, $\mathbf{QBS}(X, Y)$ comprises the measurable functions, so **QBS** can be thought of as a conservative extension of the universe of standard Borel spaces. The three conditions on quasi-Borel spaces ensure that coproducts and products of standard Borel spaces retain their universal properties in **QBS**. In fact, the category of **QBS**s has all limits and colimits. It is also cartesian closed; e.g., $\mathbb{R}^{\mathbb{R}} \coloneqq \mathbf{QBS}(\mathbb{R}, \mathbb{R})$, and $M_{(\mathbb{R}^{\mathbb{R}})} = \left\{\alpha : \mathfrak{R} \to (\mathbb{R}^{\mathbb{R}}) \;\middle|\; \mathsf{uncurry}(\alpha) \in \mathbf{QBS}(\mathfrak{R} \times \mathbb{R} \to \mathbb{R})\right\}$. For any QBS $X$, $M_X = \mathbf{QBS}(\mathfrak{R}, X)$.

*4.3.3 A Monad of Measures.* The following development is novel.

*Definition 4.7.* A *measure* $\mu$ on a quasi-Borel space is a triple $(\Omega, \alpha, \mu)$ where $\Omega$ is a standard Borel space, $\alpha \in \mathbf{QBS}(\Omega, X)$, and $\mu$ is a $\sigma$-finite measure on $\Omega$.

For example, $\Omega$ might be $\overline{\mathbb{R}}^n$ and $\mu$ might be the Lebesgue measure. A measure determines an integration operator: if $f \in \mathbf{QBS}(X, \overline{\mathbb{R}}_+)$ then define

$$\int f \,\mathrm{d}(\Omega, \alpha, \mu) \coloneqq \int_\Omega f(\alpha(x))\, \mu(\mathrm{d}x)$$

using Lebesgue integration according to $\mu$. We say that two measures are equal, denoted $(\Omega, \alpha, \mu) \approx (\Omega', \alpha', \mu')$, if they determine the same integration operator. We write $[\Omega, \alpha, \mu]$ for an equivalence class of measures.

As an aside, we note that not every integration operator on $\overline{\mathbb{R}}$ in the classical sense is a measure in the sense of Def. 4.7, because we restrict to $\sigma$-finite $\mu$. Technically, the only integration operators that arise in this way are those corresponding to s-finite measures. This is a class of measures that includes the probability measures, and which works well with iterated integration and probabilistic programming [Staton 2017].

The measures up-to $\approx$ form a monad, as follows. First, the set of all measures $\mathrm{M}\,X$ forms a QBS by setting $M_{\mathrm{M}X} = \left\{\lambda r.\, [D_r, \alpha(r, -), \mu|_{D_r}] \;\middle|\; \mu\, \sigma\text{-finite on } \Omega,\ D \subseteq \mathfrak{R} \times \Omega \text{ measurable},\ \alpha \in \mathbf{QBS}(D, X)\right\}$,

where $D_r = \{\omega \mid (r, \omega) \in D\}$. In consequence, when $\Omega'$ is a standard Borel space, for every morphism $f : \Omega' \to \mathrm{M}\, X$, there exist $\Omega$, $\mu$, $D \subseteq \Omega' \times \Omega$ and $\alpha \in \mathbf{QBS}(D, X)$ such that $f(\omega') = [D_{\omega'}, \alpha(\omega', -), \mu|_{D_{\omega'}}]$. One intuition is that $\alpha$ is a partial function $\Omega' \times \Omega \to X$, with domain $D$.

The unit of the monad, return $: X \to \mathrm{M}\, X$, is $\mathrm{return}(x) \coloneqq [\mathbb{1}, \lambda\_.\, x, \delta_{()}]$ where $\delta_{()}$ is the Dirac measure on the one-point space $\mathbb{1}$. We often write $\underline{\delta}_x$ for $\mathrm{return}(x)$. The bind $\gg\!=\!: \mathrm{M}\, X \times \mathrm{M}\, Y^X \to \mathrm{M}\, Y$ is

$$[\Omega, \alpha, \mu] \gg\!= \varphi \;\coloneqq\; [D, \beta, (\mu \otimes \mu')|_D]$$

where $\varphi(\alpha(r)) = [D_r, \beta(r, -), \mu']$. Note that $(\varphi \circ \alpha) : \Omega \to \mathrm{M}\, X$ must be of this form because it is a morphism from a standard Borel space. The measure $\mu \otimes \mu'$ is the product measure, which exists because $\mu$ and $\mu'$ are $\sigma$-finite.

This structure satisfies the monad laws, it is commutative by the Fubini-Tonelli theorem, and it satisfies the biproduct axioms, and so it is a model of synthetic measure theory. Every measure on $\mathbb{1}$ is equivalent to one of the form $([0, r], !, \mu)$ where $r \in \overline{\mathbb{R}}_+$, $! : [0, r] \to \mathbb{1}$ is the unique such random element, and $\mu$ is the Lebesgue measure. Thus $\mathrm{M}\, \mathbb{1} \cong \overline{\mathbb{R}}_+$.

As another aside, we note that when $\Omega, \Omega'$ are standard Borel spaces, the Kleisli morphisms $\Omega \to \mathrm{M}\, \Omega'$ correspond to s-finite kernels, which were shown in [Staton 2017] to provide a fully complete model of first-order probabilistic programming.

## 5 CONTINUOUS INFERENCE

We now develop the continuous counterpart to Sec. 3. The semantic structure of the category of quasi-Borel spaces allows us to transport many of the definitions with little change. For example, a monadic interface $\underline{T}$ consists of analogous data, but the assignments are indexed by quasi-Borel spaces, $T$ assigns quasi-Borel spaces, and $\underline{\mathrm{return}}^{\underline{T}}$ and $\gg\!=\!^{\underline{T}}$ assign quasi-Borel space morphisms.

*Definition 5.1.* A *continuous representation* $\underline{T}$ is a tuple $(T, \underline{\mathrm{return}}^{\underline{T}}, \gg\!=\!^{\underline{T}}, m^{\underline{T}})$ consisting of:

- a monadic interface $(T, \underline{\mathrm{return}}^{\underline{T}}, \gg\!=\!^{\underline{T}})$;
- an assignment of a *meaning* morphism $m^{\underline{T}}_X : T\, X \to \mathrm{M}\, X$ for every space $X$

such that $m^{\underline{T}}$ preserves $\underline{\mathrm{return}}^{\underline{T}}$ and $\gg\!=\!^{\underline{T}}$.

A *sampling* representation is a tuple $(T, \underline{\mathrm{return}}^{\underline{T}}, \gg\!=\!^{\underline{T}}, m^{\underline{T}}, \underline{\mathbf{sample}}^{\underline{T}})$ such that its first four components form a continuous representation, it has an additional **Qbs**-morphism $\underline{\mathbf{sample}}^{\underline{T}} : \mathbb{1} \to T\, \mathbb{I}$, and $m^{\underline{T}}$ maps $\underline{\mathbf{sample}}^{\underline{T}}()$ to the uniform **Qbs**-measure $\mathbf{U} = [\mathbb{I}, \mathrm{id}, \mathsf{Uniform}]$ on the unit interval $\mathbb{I}$, where $\mathsf{Uniform}$ is the usual uniform distribution on $\mathbb{I}$.

A *conditioning* representation $\underline{T}$ is similarly a tuple $(T, \underline{\mathrm{return}}^{\underline{T}}, \gg\!=\!^{\underline{T}}, \underline{\mathrm{score}}^{\underline{T}}, m^{\underline{T}})$, with a **Qbs**-morphism $\underline{\mathrm{score}}^{\underline{T}} : \mathbb{R}_+ \to T\, \mathbb{1}$ such that for each $r$, $m^{\underline{T}}$ maps $\underline{\mathrm{score}}^{\underline{T}}(r)$ to the $r$-rescaled unit **Qbs**-measure $r \odot \underline{\delta}_{()} = [\mathbb{1}, \lambda\_.\, (), r \cdot \delta_{()}]$.

An *inference* representation $\underline{T}$ is a tuple $(T, \underline{\mathrm{return}}^{\underline{T}}, \gg\!=\!^{\underline{T}}, \underline{\mathbf{sample}}^{\underline{T}}, \underline{\mathrm{score}}^{\underline{T}}, m^{\underline{T}})$ with the appropriate components forming both a sampling representation and a conditioning representation.

This definition refines Def. 3.2 with sampling and scoring representations, allowing us to talk about inference transformers that augment a representation of one kind into another.

*Example 5.2 (Continuous sampler).* By analogy with Ex. 3.3, we define in Fig. 8a a sampling representation using the type $\mathsf{Sam}\, \alpha \coloneqq \{\mathsf{Return}\, \alpha \mid \mathsf{Sample}\, (\mathbb{I} \to \mathsf{Sam}\, \alpha)\}$. Validating the preservation of sample and the monadic interface is straightforward. It also follows from more general principles: $\underline{\mathsf{Sam}}$ is the initial monad with an operation $\mathbf{sample} : T\, \mathbb{I}$.

We define inference transformations between any two representations as in Def. 3.5. We have four kinds of representations, and when defining transformers we can augment a representation with additional capabilities:

**instance** *Sampling Monad* (Sam) **where**
  **return** $x$ = Return $x$
  $a \gg\!= f$ = **match** $a$ **with** {
    Return $x \to f(x)$
    Sample $k \to$
      Sample $(\lambda r.\, k(r) \gg\!= f)$}
  sample = Sample $\lambda r.$ (Return $r$)
  $m\, a$ = **match** $a$ **with** {
    Return $x \to \delta_x$
    Sample $k \to \int_{\mathbb{I}} k(x)\mathrm{U}(\mathrm{d}x)$}

(a) Continuous sampler representation

**instance** *Cond Trans* (W) **where**
  $\mathbf{return}_{\mathrm{W}\,\underline{T}}\, x = \mathrm{return}^{\underline{T}}(1, x)$
  $a \gg\!=_{\mathrm{W}\,\underline{T}}\, f$ = $\underline{T}$.**do** $\{(r, x) \leftarrow a;$
    $(s, y) \leftarrow f(x);$
    **return**$(r \cdot s, y)\}$
  $(\mathbf{tmap}\, \underline{t})_X = t_{\mathbb{R}_+ * X}$
  $\mathbf{lift}_{\underline{T}}\, a = \underline{T}$.**do** $\{x \leftarrow a; \mathbf{return}(1, x)\}$
  $m_{\mathrm{W}\,\underline{T}}\, a = \lambda x.\ \int_{\mathbb{R}_+ \times X} r \odot \delta_x\, m^{\underline{T}}(a)(\mathrm{d}r, \mathrm{d}x)$
  $\mathbf{score}_{\mathrm{W}\,\underline{T}}\, r = \mathrm{return}^{\underline{T}}(r, ())$

(b) Continuous weighting inference transformer

Fig. 8. Continuous representations and tranformers

*Definition 5.3.* Let $k_1, k_2$ be a pair of kinds of representation. A $k_1$ *to* $k_2$ *transformer* $\underline{F}$ is a tuple $(F, \mathrm{tmap}^{\underline{F}}, \mathrm{lift}^{\underline{F}})$ consisting of an assignments of:

- a $k_2$ representation $F\,\underline{T}$ to every $k_1$ representation $\underline{T}$;
- an inference transformation $\mathrm{tmap}^{\underline{F}}\, \underline{t} : \underline{FT} \to \underline{FS}$ to every transformation $\underline{t} : \underline{T} \to \underline{S}$; and
- an inference transformation $\mathrm{lift}_T : \underline{T} \to F\,\underline{T}$ to every $k_1$ representation $\underline{T}$.

When the two kinds $k_1, k_2$ differ, we say that that the transformer is *augmenting*.

When defining a $k_1$ to $k_2$ transformer, we adopt a Haskell-like type-class constraint notation $k_1 \implies k_2$ used for example in Fig. 11a.

*Example 5.4.* By analogy with Ex. 3.9, Fig. 8b presents the *continuous weighting* transformer structure on $\mathrm{W}\,\underline{T}\,X \coloneqq T(\mathbb{R}_+ * X)$. It augments any representation transformer with conditioning capabilities. Each conditioning operation is deferred to the return value, and so we can view this transformer as freely adding a conditioning operation that commutes with all other operations. When the starting representation had conditioning capabilities, we have an inference transformation $\mathrm{waggr} : \mathrm{W}\,\underline{T} \to \underline{T}$, given by $\mathrm{waggr}\, a \coloneqq \underline{T}$.**do** $\{(r, x) \leftarrow a; \mathrm{score}^{\underline{T}}\, r; \mathbf{return}\, x\}$ which conditions based on the aggregated weight.

Its validity follows from a straightforward calculation using the meaning preservation of $\underline{T}$.

In the continuous case, the output of the final inference transformation will always be W Sam $X$ or a similar Pop Sam $X$ described in the next section. From this representation, we obtain the Monte Carlo approximation to the posterior by using a random number generator to supply the values required by Sam. Interpreting the program directly in W Sam $X$ and sampling from that would correspond to simple importance sampling from the prior, which usually needs a very large number of samples to give a good approximation to the posterior. Our goal in approximate Bayesian inference is therefore to find another representation for the program and a sequence of inference transformations that map it to W Sam $X$. While, in principle, this output represents the same posterior distribution, hopefully it uses a representation that requires fewer samples to obtain a good approximation than a direct interpretation in W Sam $X$. We emphasise that approximation is only done in this final sampling step, while all the inference transformations that happen before it are exact.

## 6 SEQUENTIAL MONTE CARLO

*Sequential Monte Carlo (SMC)* is a family of inference algorithms for approximating sequences of distributions. In SMC, every distribution is approximated by a collection of weighted samples called a *population*, with each population obtained from the previous one in the sequence by a suitable transformation. In a sub-class of SMC known as *particle filters*, each distribution in the sequence is generated by a known random process from the previous distribution and we can apply this process to samples in the previous population to obtain the current population. In particle filters the samples in the population are called *particles*.

A common problem with particle filters is that, after multiple steps, a few particles have much larger weights than the remaining ones, effectively reducing the sample size in the population well below the actual number of particles, a phenomenon known as *sample impoverishment*. To ameliorate this problem, particle filters introduce additional *resample* operations after each step in the sequence, which constructs a new population by sampling with replacement from the old one. The new population has uniform weights across its particles. In the setting of probabilistic programming, we use suspended computations as particles, and their associated weight is their currently accumulated likelihood.

We show how to decompose a particle filter into a stack of two transformers: a representation to conditioning transformer for representing a population of particles, and a conditioning to conditioning transformer that allows us to run a particle until its next conditioning operation. We define each step of the SMC algorithm as an inference transformation on this stack. We can then apply this stack of transformers to a sampling representation to obtain a correct by construction variation of SMC. The algorithm we obtain is known as the particle filter with multinomial resampling [Doucet and Johansen 2011] that uses the prior as the proposal distribution, but throughout this paper we refer to it simply as SMC.

### 6.1 The Population Transformer

Given a representation $\underline{T}$, we define a representation structure over $\mathsf{Pop}\,\underline{T}\,X := T(\mathsf{List}(\mathbb{R}_+ * X))$. We further deconstruct this representation transformer as the composition of two transformers: the continuous weighting transformer W from Ex. 5.4, and Haskell's notorious ListT transformer.

The negative reputation associated to the transformer $\mathsf{ListT}\,\underline{T}\,X := T(\mathsf{List}\,X)$ stems from its failure to validate the monad laws when $\underline{T}$ is not commutative.[2] However, it is a perfectly valid representation transformer, described in Fig. 9a, since we do not require that representations satisfy monad laws.

To prove the meaning function preserves return, simply calculate. For $\gg=$ preservation, show:

$$a_s : \mathsf{List}(T\,X) \vdash m^{\mathsf{ListT}\,\underline{T}}(\mathsf{sequence}\,a_s) = \sum_{a \in a_s} m^{\underline{T}}(a)$$

and proceed via straightforward calculation using the linearity of the Kock integral and the commutative ($\sigma$-)monoid structure on measures.

By composing the two representation transformers, we obtain the representation to conditioning transformer Pop, given explicitly in Fig. 9b.

Fig. 9c presents a $\mathbb{N}_+$-indexed family of inference transformations. Fix any $n \in \mathbb{N}$. The spark function generates a population of particles with the unit value, and the same weight $\frac{1}{n}$. Thus, spawn $(n, a)$ takes a distribution $a$ over particle populations, sparks $n$ equally weighted particles, and for each of them, samples a population based on $a$. A straightforward calculation confirms that the meaning of spark is 1, and so $\mathsf{spawn}\,(n, -) : \mathsf{Pop}\,\underline{T} \to \mathsf{Pop}\,\underline{T}$ is an inference transformation. In

[2] For a list transformer "done right", see Jaskelioff's thesis [2009], and its generalisations [Fiore and Saville 2017; Piróg 2016].

Auxiliary functions:
$\text{sequence} : \text{List}(TX) \to T(\text{List}\, X)$
$\text{sequence} \coloneqq \text{foldr}\,(\text{return}[\,])$
$\qquad (\lambda(a, r).\ \underline{T}.\mathbf{do}\,\{x \leftarrow a;$
$\qquad\qquad x_s \leftarrow r;$
$\qquad\qquad \mathbf{return}(x :: x_s)\}$
$\text{concat} : \text{List}(\text{List}\, X) \to \text{List}\, X$
$\text{concat} \coloneqq \text{foldr}\,[\,] +\!\!+$
$\sum_{x \in x_s} f(x) \coloneqq \text{foldr}\, 0\ \left(\lambda(x, s).\ f(x) + s\right)\ x_s$

**instance** *Rep Trans* (ListT) **where**
$\mathbf{return}_{\text{ListT}\,\underline{T}}\, x = \text{return}^{\underline{T}}[x]$
$a \gg\!=_{\text{ListT}\,\underline{T}}\, f = \underline{T}.\mathbf{do}\,\{x_s \leftarrow a;$
$\qquad\qquad \mathbf{let}\, b_s = \text{map}\, f\ x_s\ \mathbf{in}$
$\qquad\qquad y_{ss} \leftarrow \text{sequence}\, b_s;$
$\qquad\qquad \mathbf{return}(\text{concat}\, y_{ss})\}$
$m_{\text{ListT}\,\underline{T}}\, a = \oint_{\text{List}\, X} m^{\underline{T}}(a)(\mathrm{d}x_s) \sum_{x \in x_s} \underline{\delta}_x$
$\mathbf{lift}_{\text{ListT}\,\underline{T}}\, a = \underline{T}.\mathbf{do}\,\{x \leftarrow a; \mathbf{return}\,[x]\}$
$(\mathbf{tmap}\, \underline{t})_X = \underline{t}_{\text{List}\, X}$

(a) The list transformer

**instance** *Cond Trans* (Pop) **where**
$\mathbf{return}_{\text{Pop}\,\underline{T}} = \mathbf{return}_{(\mathsf{W} \circ \text{ListT})\underline{T}}$
$\gg\!=_{\text{Pop}\,\underline{T}} = \gg\!=_{(\mathsf{W} \circ \text{ListT})\underline{T}}$
$\mathbf{lift}_{\text{Pop}\,\underline{T}} = \mathbf{lift}_{\mathsf{W}(\text{ListT}\,\underline{T})} \circ \mathbf{lift}_{\text{ListT}\,\underline{T}}$
$\mathbf{tmap}_{\text{Pop}\,\underline{T}} = \mathbf{tmap}_{\mathsf{W}(\text{ListT}\,\underline{T})} \circ \mathbf{tmap}_{\text{ListT}\,\underline{T}}$
$m_{\text{Pop}\,\underline{T}} = m_{(\mathsf{W} \circ \text{ListT})\underline{T}}$
$= \lambda a.\ \oint_{\text{List}(\mathbb{R}_+ \times X)} m^{T}(a)(\mathrm{d}x_s) \sum_{(r,x) \in x_s} r \odot \underline{\delta}_x$
$\mathbf{score}_{\text{Pop}\,\underline{T}} = \mathbf{score}_{(\mathsf{W} \circ \text{ListT})\underline{T}}$

(b) The population transformer

$\text{replicate} : \mathbb{N} * X \to \text{List}\, X$
$\text{replicate}(n, x) \coloneqq \mathbb{N}.\mathbf{fold}\,\lambda\{\ \text{Zero} \to [\,]$
$\qquad\qquad \mid \text{Succ}\, x_s \to x :: x_s\}\ n$
$\text{spark} : \mathbb{N}_+ \to \text{Pop}\,\underline{T}\,\mathbf{1}$
$\text{spark} \coloneqq \text{return}^{\underline{T}}\left(\text{replicate}(n, (\tfrac{1}{n}, ()))\right)$
$\text{spawn} : \mathbb{N}_+ * \text{Pop}\,\underline{T}\,X \to \text{Pop}\,\underline{T}\,X$
$\text{spawn}(n, a) \coloneqq \text{Pop}\,\underline{T}.\mathbf{do}\,\{\text{spark}\, n; a\}$

(c) Spawning new particles

Fig. 9. Representing populations

the version of SMC we consider below, we will only pass to spawn a distribution $a$ over uniformly-weighted single-particle populations.

We use spawn to *resample* a new population. Thinking operationally, we have a population of weighted particles and we obtain a new population by sampling with replacement from the current one, where the probability of selecting a given particle is proportional to its weight. Doing so is equivalent to simulating a discrete weighted sample using a uniform one.

Lemma 6.1. *There is a* **Qbs**-*morphism* $\text{dwrand} : \text{List}(\mathbb{R}_+ * X) * \mathbb{I} \to \{\text{Take}\, X \mid \text{Fail}\}$ *such that:*

- *For all* $x_s$ *for which* $\sum_{(r,\_) \in x_s} r = 0$, *we have* $\text{dwrand}(x_s, -)_* \mathrm{U} = \underline{\delta}_{\text{Fail}}$.
- *For all* $x_s$ *for which* $w \coloneqq \sum_{(r,\_) \in x_s} r > 0$, *we have* $\text{dwrand}(x_s, -)_* \mathrm{U} = \sum_{(r,x) \in x_s} \frac{r_i}{w} \odot \underline{\delta}_{\text{Take}\, x}$.

Fig. 10a presents one such morphism, though its precise implementation does not matter to our development. As a consequence, for every sampling representation $\underline{T}$ for which we have an element $\text{fail} : TX$ such that $m^{\underline{T}}(\text{fail}) = 0$, we can define a discrete weighted sampler $\text{dwsample}^{\underline{T}}(x_s) : \text{List}(\overline{\mathbb{R}}_+ X) \to TX$ in Fig. 10b which will then satisfy $m^{\underline{T}}(\text{dwsample}^{\underline{T}}(x_s)) = \sum_{(r,x) \in x_s} r \odot \underline{\delta}_x$.

The resampling step in Fig. 10c operationally takes the current population, creates a computation/thunk that samples a single particle from this population, and then spawns $n$ new particles that are initialised with this thunk. The morphism $\text{resample}\,(n, -) : \text{Pop}\,\underline{T} \to \text{Pop}\,\underline{T}$ is an inference transformation because, as we know, $\text{spawn}\,(n, -)$ is one and $\text{dwsample}^{\text{Pop}\,\underline{T}} : \text{Pop}\,\underline{T} \to \text{Pop}\,\underline{T}$ samples a population consisting of just a single unit weight particle with a probability proportional to its renormalised weight in the original population.

```
dwrand(x_s, r) :=
  let w = ∑_{(r,_)∈x_s} r in
  if   w = 0
  then Fail
  else foldr (w · r, Fail)
         (λ((s, x), (fuel, result)).
         if   0 ≤ fuel < s
         then(−1, Take x)
         else −− potential underflow
              (fuel − s, result))
         x_s
```

(a) A discrete weighted randomiser

```
dwsample^T(x_s) :=
  T.do {score(∑_{(r,_)∈x_s} r);
        r ← sample;
        match dwrand (x_s, r) with {
              Fail   →fail
            | Take x→return x}}
```

(b) A discrete weighted sampler

```
resample : ℕ+ ∗ Pop T X → Pop T X
resample(n, a) :=
    T.do {x_s ← a;
          spawn(n, dwsample^{Pop T} x_s)}
```

(c) Resampling

Fig. 10. The resampling transformation

```
instance Cond ⟹ Cond Trans(Sus) where
  return_{Sus T} x   = return_T(Return x)
  a ≫=_{Sus T} f     = fold (λb. T.do {
                         t ← b;
                         match t with {
                               Return x→f(x)
                             | Yield c   →Yield c})}
                         a
  lift_{Sus T} a     = T.do {x ← a; return_{Sus T} x}
  (tmap_{Sus T} t)_X = Sus TX.fold (λb. m_S(b))
  m_{Sus T} a        = m_T(finish_{Sus T}(a))
  score r            = return_T(Yield lift_{Sus T}(score r))
```

(a) The suspension transformer

```
advance_T  : Sus TX → Sus TX
advance_T a = T.do {
                t ← a;
                match t with {
                      Return x→return_T x
                    | Yield t   →t}}
finish_T   : Sus TX → TX
finish_T a  = fold λb. T.do {
                t ← b;
                match t with {
                      Return x→return x
                    | Yield b   →b}}
```

(b) Suspension operations

Fig. 11. The suspension transformation

### 6.2 The Suspension Transformer

The second transformer in the SMC algorithm allows us to suspend computation after each conditioning. The suspension transformer equips the standard *resumption* monad transformer $\mathrm{Sus}\,\underline{T}\,X := T\{\mathrm{Return}\,X \mid \mathrm{Yield}\,(\mathrm{Sus}\,\underline{T}\,X)\}$, presented in Fig. 11a, with inference transformations.

The two transformations on suspended computations in Fig. 11b take one step, and complete the computation, accordingly. As the meaning function for the transformed representation returns the meaning the computation would have if it was allowed to run to completion, these two operations do not change the meaning and so form inference transformations.

We can now put all the components together:

Theorem 6.2. *Let $\underline{T}$ be a sampling representation. For every pair of natural numbers $n$, $k$, the following composite forms an inference transformation:*

$$\mathrm{smc}^{\underline{T}}_{n,k} := (\mathsf{Sus} \circ \mathsf{Pop})\underline{T} \xrightarrow{\mathbf{tmap}_{\mathsf{Sus}}\ \mathrm{spawn}(n,-)} (\mathsf{Sus} \circ \mathsf{Pop})\underline{T} \xrightarrow{(\mathrm{advance} \circ \mathbf{tmap}_{\mathsf{Sus}}\ \mathrm{resample}(n,-))^{\circ k}} (\mathsf{Sus} \circ \mathsf{Pop})\underline{T} \xrightarrow{\mathrm{finish}} \mathsf{Pop}\,\underline{T}$$

In the above $(-)^{\circ -} : X^X \times \mathbb{N} \to X^X$ denotes $n$-fold composition. The transformation $\mathrm{smc}^{\underline{T}}_{n,k}$ amounts to running the SMC algorithm with $n$ particles for $k$ steps. If the representation $\underline{T}$ is operational in nature, such as the continuous sampler Sam, we get a sequence of weighted values over the return type when we run the resulting representation. By construction, the distribution on the results, rescaled according to their final weights, would be identical to the desired posterior distribution.

When the representation $\underline{T}$ is not a commutative monad, like the continuous sampler Sam, the resulting representation $\mathsf{Pop}\,\underline{T}$ is not a monad: the monad laws do not hold. Therefore, to encompass representations of $\mathsf{Pop}\,\underline{T}$ one must generalise beyond monads.

## 7 TRACE MARKOV CHAIN MONTE CARLO

Markov Chain Monte Carlo (MCMC) algorithms operate by repeatedly using a transition kernel to generate a new sample from a current one. Thus they can be thought of as performing a random walk around the space they are exploring. If the transition kernel is well-behaved, they are guaranteed to preserve the distribution. A popular MCMC algorithm used for Bayesian inference is Metropolis-Hastings (MH), where the transition kernel consists of a proposal kernel followed by a decision to either accept the proposed sample or keep the old one. The accept or reject step is used to correct for bias introduced by the proposal kernel, thus producing a valid MCMC algorithm for a rich family of proposal kernels.

MH is a general inference method, but it requires specialised knowledge about the space on which they operate on. In the context of a probabilistic programming language, the *Trace MH* algorithm replaces the unknown target space with the space of program *traces*, which are shared by all probabilistic programs. Thus, Trace MH allows probabilistic programming language designers to devise general-purpose kernels to effectively explore traces.

We analyse the the Trace MH as follows. First, we prove a quasi-Borel space counterpart of the *Metropolis-Hastings-Green (MHG) Theorem*, that forms the theoretical foundation for the correctness of MH. We then present the *tracing* representation and show its validity. We present the Trace MH algorithm, parameterised by a proposal kernel for traces, and give sufficient conditions on this kernel for the resulting transformation to be valid. We then give a concrete proposal kernel and show that it satisfies these general conditions.

### 7.1 Abstract Metropolis-Hastings-Green

In the abstract, the key ingredient in MH is the *Metropolis-Hastings-Green (MHG)* morphism $\eta$ presented in Fig. 12a, formulated in terms of an arbitrary inference representation $\underline{T}$. This transformation is usually known as the *update step* of the MH algorithm. It is parameterised by a (representation of a) *proposal* kernel $\psi : X \to T\,X$, and by a chosen (representation of a) Radon-Nikodym derivative $\rho : X \times X \to \overline{\mathbb{R}}_+$.

To use $\eta$ in an inference transformation, we need to provide well-behaved parameters $\psi$, $\rho$, and their behaviour may depend on the representation of the input distribution $a$. In particular, the parameter $\rho$ should represent a well-behaved appropriate Radon-Nikodym derivative. To simplify our proofs, we also require that the proposal kernel $\psi$ is Markov, which suffices for our application.

$$\psi : (T\,X)^X, \rho : \overline{\mathbb{R}}_+^{X\times X}, a : T\,X \vdash$$
$$\underline{T}.\mathbf{do}\ \{x \leftarrow a;$$
$$\quad y \leftarrow \psi(x);$$
$$\quad r \leftarrow \mathrm{sample};$$
$$\quad \mathbf{if}\ r < \min(1, \rho(x,y))$$
$$\quad \mathbf{then\ return}\ y$$
$$\quad \mathbf{else\ return}\ x\}$$

(a) The Metropolis-Hastings-Green morphism $\eta_{\psi,\rho}(a)$

$$p \in t = \mathbf{match}\,(p,t)\,\mathbf{with}\ \{([\,], \mathrm{Return}\,x) \to \mathrm{True}$$
$$(r :: r_s, \mathrm{Sample}\,f) \to [r_s \in f(r)]$$
$$-- \text{ any other case:}$$
$$(\_, \_) \to \mathrm{False}\}$$

$$w_- : \textstyle\sum_{t\in \mathsf{WSam}\,X} \mathsf{Paths}\,t \to \mathbb{R}_+ \qquad v_- : \textstyle\sum_{t\in \mathsf{WSam}\,X} \mathsf{Paths}\,t \to X$$
$$w_{\mathrm{Return}\,(r,x)}([\,]) = r \qquad v_{\mathrm{Return}\,(r,x)}([\,]) = x$$
$$w_{\mathrm{Sample}\,t_-}(s :: r_s) = w_{t_s}(r_s) \qquad v_{\mathrm{Sample}\,t_-}(s :: r_s) = v_{t_s}(r_s)$$

(b) Traces through a probabilistic program

Fig. 12. Basic notions in Trace MH

Theorem 7.1 (Metropolis-Hastings-Green). *Let $X$ be a qbs, $a \in TX$ a distribution, $\psi : X \to TX$ a kernel, and $\rho : X \times X \to \overline{\mathbb{R}}_+$ a* **Qbs***-morphism. Set $k := m^{\underline{T}} \circ \psi$ and $\underline{\mu} := [\rho \neq 0] \odot (m^{\underline{T}}(a) \boxtimes k)$.*

*Assume that: (1) $k$ is Markov; (2) $[1 = (\rho \circ \mathrm{swap}) \cdot \rho]$ holds $\underline{\mu}$-a.e.; (3) $\rho$ is a Radon-Nikodym derivative of $\mathrm{swap}_*\underline{\mu}$ with respect to $\underline{\mu}$; and (4) $\rho(x,y) = 0 \iff \rho(y,x) = 0$ for all $x, y \in X$.*

*Then $(m^{\underline{T}} \circ \eta_{\psi,\rho})(a) = m^{\underline{T}}(a)$.*

Using Kock's synthetic measure theory, we were able to follow closely standard measure-theoretic proofs of MHG [Geyer 2011]. The synthetic setting highlights the different roles each of the three abstractions: a.e.-equality, a.e.-properties, and Radon-Nykodim derivatives play in the proof that our formulation exposes (cf. § 4.2.3).

### 7.2 Tracing Representation

A sampling *trace* is a sequence of samples that occur during the execution of a probabilistic program. We represent such programs as elements of the continuous weighted sampler $\mathsf{W\,Sam}$ from (cf. Fig. 8). Consequently, the collection of traces through a program $t \in \mathsf{W\,Sam}\,X$ is a subset of $\mathsf{List}\,\mathbb{I}$. Fig. 12b defines a measurable predicate $[\in] : \mathsf{W\,Sam}\,X \times \mathsf{List}\,\mathbb{I} \to \mathsf{bool}$ that tests whether a given sequence $p$ of probabilistic choice forms a complete trace in the program $t$. Consequently, we can define the set of *paths* through a given program $t$ by $\mathsf{Paths}\,t := \{p \in \mathsf{List}\,\mathbb{I} \,|\, p \in t\} \subseteq \mathsf{List}\,\mathbb{I}$, and equip it with the subspace structure it inherits from $\mathsf{List}\,\mathbb{I}$. We can therefore define the set:

$$\sum_{t\in \mathsf{WSam}\,X} \mathsf{Paths}\,t := \{(t,p) \in \mathsf{WSam}\,X \times \mathsf{List}\,\mathbb{I} \,|\, p \in t\} \subseteq \mathsf{WSam}\,X \times \mathsf{List}\,\mathbb{I}$$

which we can also equip with a subspace structure. We can now define the *weight* $w_-$ and *valuation* $v_-$ morphisms in Fig. 12b that retrieve the likelihood and value at the end of a trace.

We can now define the tracing inference representation. It is parameterised by an inference representation $\underline{T}$ and given for $X$ as the following subspace of $\mathsf{WSam}\,X \times T(\mathsf{List}\,\mathbb{I})$:

$$\mathrm{Tr}\,\underline{T}\,X := \left\{ (t,a) \in \mathsf{WSam}\,X \times T(\mathsf{List}\,\mathbb{I}) \,\middle|\, \begin{array}{l} [\in t]\ m_{\underline{T}}(a)\text{-a.e., and} \\ m_{\mathsf{WSam}}(t) = \int_{\mathsf{List}\,\mathbb{I}} \underline{\delta}_{v_t(p)}\, m_{\underline{T}}(a)(\mathrm{d}p) \end{array} \right\}.$$

Thus, a representation consists of a program representation $t$, together with a distribution $a$ on all lists, but maintaining two invariants. First, the lists are $m_{\underline{T}}(a)$-almost-everywhere paths through $t$, and so we can indeed think of $a$ as a representation of a distribution over traces. Second, if we calculate the posterior of the paths through $t$ according to $m_{\underline{T}}(a)$, it should have the same meaning as the original program.

$$
\begin{aligned}
&\textbf{instance } \mathit{Inf} \implies \mathit{Inf}\ \mathit{Monad}\,(\mathrm{Tr}\,\underline{T})\ \textbf{where}\\
&\quad \textbf{return}\, x \quad = (\mathrm{return}_{\mathsf{WSam}}\, x, \mathrm{return}_{\underline{T}}[\,])\\
&\quad (t,a) \gg\!= (f,g) = (t \gg\!=_{\mathsf{WSam}} f, \underline{T}.\textbf{do}\,\{p \leftarrow a;\\
&\qquad\qquad\qquad\qquad\qquad\qquad q \leftarrow g \circ \upsilon_t(p);\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}(p +\!\!+ q)\})\\
&\quad m(t,a) \quad = m_{\mathsf{WSam}}(t) = \oint_{\mathsf{List}\,\mathbb{I}} \underline{\delta}_{\upsilon_t(p)}\, m_{\underline{T}}(a)(\mathrm{d}p)\\
&\quad \textbf{tmap}\,\underline{t} \quad = \mathrm{id} \times \underline{t}_{\mathsf{List}\,\mathbb{I}}\\
&\quad \mathrm{sample} \quad = (\textbf{sample}_{\mathsf{WSam}},\\
&\qquad\qquad \underline{T}.\textbf{do}\,\{r \leftarrow \mathrm{sample}; \textbf{return}[r]\})\\
&\quad \textbf{score}\, r \quad = (\textbf{score}_{\mathsf{WSam}},\\
&\qquad\qquad \underline{T}.\textbf{do}\,\{\textbf{score}\, r; \textbf{return}[\,]\})
\end{aligned}
$$

(a) The tracing inference

$$
\begin{aligned}
&\eta^{\mathrm{Tr}\,T}_{\psi,\rho} : \mathrm{Tr}\,T\,X \to \mathrm{Tr}\,T\,X\\
&\eta^{\mathrm{Tr}\,T}_{\psi,\rho}\,(t,a) \coloneqq \left(t, \eta_{\psi_t,\rho_t}(a)\right)
\end{aligned}
$$

(b) Trace MH update-step

$$
\begin{aligned}
&\mathrm{pri}_{\underline{T}} : \mathsf{WSam}\,X \to T(\mathsf{List}(\mathbb{I}))\\
&\mathrm{pri}_{\underline{T}}(t) \coloneqq \textbf{fold}\\
&\lambda\{\mathsf{Return}\,(r,x) \to \mathrm{return}_{\underline{T}}[\,]\\
&\ \ \mid \mathsf{Sample}\,k \quad \to \underline{T}.\textbf{do}\,\{\\
&\qquad\qquad\qquad r \leftarrow \textbf{sample}_{\underline{T}};\\
&\qquad\qquad\qquad k(r)\}\}
\end{aligned}
$$

(c) Prior representation

Fig. 13. Building blocks of Trace MH

We stress that an implementation need *not* compute the meaning of the program. But this representation guarantees that the meaning will be preserved by the inference operations.

Note that the integrand in the definition of $(t,a) \in \mathrm{Tr}\,T\,X$ is only partially defined. This partiality is not an issue because the first condition guarantees it is $m^T(a)$-a.e. defined. We can then choose the constantly 0 distribution when $p \notin t$.

Fig. 13a presents the inference representation structure of $\mathrm{Tr}\,\underline{T}$. Most of the proof revolves around preseving the invariant, i.e., that these definitions define set-theoretic functions.

The inference transformation $\mathrm{marginal}_{\underline{T}} : \mathrm{Tr}\,\underline{T}X \to \underline{T}X$ *marginalises* the trace transformer once it is no longer useful. It first samples a path and then uses it to run the program discarding the weight: $\mathrm{marginal}\,(t,a) = \textbf{do}\,\{x \leftarrow a; \textbf{return}\ \upsilon_t(x)\}$. Its correctness is precisely the invariant.

### 7.3 Inference with MHG

The transition from $\underline{T}$ to $\mathrm{Tr}\,\underline{T}$ still requires a proposal kernel and a representation of the appropriate derivative, but these can now be given in terms of concrete traces.

Given an inference representation $\underline{T}$, a *trace proposal kernel* is a transformation representing a kernel $\psi : \left(\sum_{t \in \mathsf{WSam}\,X} \mathrm{Paths}\,t\right) \to T(\mathrm{List}\,\mathbb{I})$. A *trace derivative* is a transformation representing the derivative $\rho : \left(\sum_{t \in \mathsf{WSam}\,X} \mathrm{Paths}\,t \times \mathrm{Paths}\,t\right) \to \overline{\mathbb{R}}_+$. Given a trace proposal kernel $\psi$ and a trace derivative $\rho$, Fig. 13b presents the trace MHG update transformation using the corresponding MHG update on $T(\mathrm{List}\,\mathbb{I})$.

The Trace MH update step requires some assumptions to form an inference transformation:

Theorem 7.2 (Trace Metropolis-Hastings-Green). *Let $T$ be an inference representation, $\psi$ a trace proposal kernel, and $\rho$ a trace derivative. Assume that, for every $(t,a) \in \mathrm{Tr}\,T\,X$, letting $k \coloneqq m^T \circ \psi_t$ and $\underline{\mu} \coloneqq [\rho_t \neq 0] \odot (m^T(a) \boxtimes k)$: (1) $k$ is Markov; (2) $[1 = \rho_t \cdot (\rho_t \circ \mathrm{swap})]$ holds $\underline{\mu}$-a.e.; (3) $\rho_t$ is a Radon-Nikodym derivative of $\mathrm{swap}_*\underline{\mu}$ with respect to $\underline{\mu}$; and (4) $\rho_t(p,q) = 0 \iff \overline{\rho_t}(q,p) = 0$ for all $p,q \in \mathsf{List}(\mathbb{I})$. Then $\eta^{\mathrm{Tr}\,T}_{\psi,\rho} : \mathrm{Tr}\,T \to \mathrm{Tr}\,T$ is a valid inference transformation.*

We will now demonstrate such a simple and generic trace proposal kernel and trace derivative that implement a MHG update step of a popular lightweight Metropolis-Hastings algorithm in several probabilistic programming language systems [Goodman et al. 2008; Goodman and Stuhlmüller 2014; Hur et al. 2015; Wood et al. 2014].

For any inference representation $\underline{T}$, Fig. 13c defines the morphism $\mathrm{pri}_{\underline{T}}$ that maps a representation $t \in \mathsf{WSam}\, X$ to its prior distribution on paths over $t$. Let $\mathrm{U_D}(n) \in \mathrm{M}(\mathbb{N})$ be the measure for the uniform discrete distribution with support $\{0, 1, \ldots, n\}$. Intuitively, it assigns a probability $\frac{1}{n+1}$ to every element in the support. It be easily defined from $\mathbf{sample}_{\mathrm{M}}$, which denotes the uniform distribution on $\mathbb{I}$, as in Lemma 6.1.

We now define our concrete proposal $\psi_t$ and derivative, a.k.a. ratio, $\rho_t$:

$$\begin{aligned}
&\psi_t : \mathsf{List}(\mathbb{I}) \to T(\mathsf{List}(\mathbb{I})) \\
&\psi_t(p) := \underline{T}.\mathbf{do}\ \{i \leftarrow \mathrm{U_D}(|p|) \\
&\qquad\qquad q \leftarrow \mathrm{pri}^T(\mathrm{sub}(t, \mathrm{take}(i, p))) \\
&\qquad\qquad \mathbf{return}(\mathrm{take}(i, p) + q)\}
\end{aligned}
\qquad\qquad
\begin{aligned}
&\rho_t : \mathsf{List}(\mathbb{I}) \times \mathsf{List}(\mathbb{I}) \to \overline{\mathbb{R}}_+ \\
&\rho_t(p, q) := \tfrac{w_t(q)\cdot(|p|+1)}{w_t(p)\cdot(|q|+1)}
\end{aligned}$$

where $\mathrm{sub}(t, x)$ selects a subterm of a given term by following the list $x$ and $\mathrm{take}(i, p)$ retrieves the $i$-th prefix of $p$. This proposal and derivative/ratio satisfy the condition in the Trace MH.

Our approach lets us combine MH updates with other inference building block. For example, recall the SMC algorithm from Section 6.2. Each time it performs resampling, multiple particles are given the same values, which results in inadequate coverage of the space, a phenomenon known as *degeneracy*. One way to ameliorate this problem is to apply multiple MH transitions to each particle after resampling in order to spread them across the space, resulting in an algorithm known as resample-move SMC [Doucet and Johansen 2011].

The implemnetation of resample-move SMC is very similar to that of SMC from Section 6.2, except we introduce an additional layer Tr between Sus and Pop:

Theorem 7.3. *Let $\underline{T}$ be a sampling representation. For every pair of natural numbers $n$, $k$, $\ell$ the following composite forms an inference trasnformation:*

$$\begin{aligned}
\mathrm{rmsmc}^{\underline{T}}_{n,k,\ell} := (\mathsf{Sus} \circ \mathsf{Tr} \circ \mathsf{Pop})\underline{T} &\xrightarrow{\mathbf{tmap}_{\mathsf{Sus}}\, \mathbf{tmap}_{\mathsf{Tr}}\, \mathrm{spawn}(n, -)} (\mathsf{Sus} \circ \mathsf{Tr} \circ \mathsf{Pop})\underline{T} \\
&\xrightarrow{(\mathrm{advance} \circ\ \mathbf{tmap}_{\mathsf{Sus}}\, \eta^{\circ \ell}\ \circ \mathbf{tmap}_{\mathsf{Sus}}\, \mathbf{tmap}_{\mathsf{Tr}}\, \mathrm{resample}(n, -))^{\circ k}} (\mathsf{Sus} \circ \mathsf{Tr} \circ \mathsf{Pop})\underline{T} \xrightarrow{\mathrm{marginal} \circ \mathrm{finish}} \mathsf{Pop}\, \underline{T}
\end{aligned}$$

In the above we apply $\ell$ MH transitions after each resampling. Our compositional correctness criterion corresponds to a known result that resample-move SMC is an unbiased importance sampler.

## 8 RELATED WORK AND CONCLUDING REMARKS

The idea of developing a programming language for machine learning and statistics is old, and was explored at least in the early 2000s [Park et al. 2005; Ramsey and Pfeffer 2002; Thomas et al. 1992] as an interesting yet niche research topic. In the past five years, however, designing such a language and building its runtime system has become an active research area, and lead to practical programming languages and libraries [Carpenter et al. 2017; Goodman et al. 2008; Goodman and Stuhlmüller 2014; Gordon et al. 2014; Mansinghka et al. 2014; Minka et al. 2014; Murray 2013; Narayanan et al. 2016; Tran et al. 2017; Wood et al. 2014]. Most of these research efforts have focussed on developing efficient inference algorithms and implementations [Kucukelbir et al. 2015; Le et al. 2017; Tran et al. 2017; Wingate and Weber 2013]. Only a smaller amount of work has been dedicated to justifying the algorithms or other runtime systems of those languages [Borgström et al. 2015; Hur et al. 2015]. Our work contributes to this less-explored line of research by providing novel denotational techniques and tools for specifying and verifying key components of inference algorithms, in particular, those for expressive higher-order probabilistic programming languages. Such specifications can then be combined to construct the correctness argument of a complex inference algorithm, as we have shown in the paper.

The idea of constructing inference algorithms by composing transformations of an intermediate representations is, to the best of our knowledge, relatively recent. In previous work with Gordon [Ścibior et al. 2015], we manipulated a free monad representation to obtain an implementation of SMC. However, we did not implement MH, did not break down SMC further into resampling and suspension, and our semantics was not compositional. Zinkov and Shan [2017] directly manipulate syntax trees of a small language Hakaru. Their semantics is only first-order and they focus on local program transformations corresponding to solving integrals analytically, which is orthogonal to our global transformations relating to sampling algorithms.

Our approach does not yet deal with two important aspects of inference. In practice, one wants *convergence* guarantees for the inference algorithm, estimating the results within an error margin after a given number of inference steps. As any purely-measure theoretic approach, ours does not express such properties. Additionally, we can not express algorithms that rely on derivatives of the density function for the program traces, such as Hamiltonian Monte Carlo or *variational* inference. Developing a theory of differentiation over quasi-Borel spaces might enable us to express such algorithms.

Another interesting direction for future work is to develop a denotational account of some probabilistic programming languages that allow users to select or compose parts of inference algorithms [Mansinghka et al. 2014; Tran et al. 2017]. The exposure of an inference algorithm in such languages breaks the usual abstraction of probabilistic programs as distributions, and causes difficulties of applying existing semantic techniques to such programs. Our more intensional semantics may be able to overcome these difficulties. Finally, it would be interesting to consider indexed or effect-annotated versions of inference representations, transformations and transformers, where indices or annotations ensure that inference components are applied to a certain type of programs. Such a refined version of our results may lead to a way of selectively applying Hamiltonian Monte Carlo or other algorithms that assume the presence of differentiable densities and a fixed length of all paths through the program.

## ACKNOWLEDGMENTS

Supported by a Royal Society University Research Fellowship, Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) No. R0190-16-2011 'Development of Vulnerability Discovery Technologies for IoT Software Security', Engineering and Physical Sciences Research Council (ESPRC) Early Career Fellowship EP/L002388/1 'Combining viewpoints in quantum theory', an EPSRC studentship and grants EP/N007387/1 'Quantum computation as a programming language' and EP/M023974/1 'Compositional higher-order model checking: logics, models, and algorithms', a Balliol College Oxford Career Development Fellowship, and a University College Oxford Junior Research Fellowship. We would like to thank Samson Abramsky, Thorsten Altenkirch, Bob Coecke, Mathieu Huot, Radha Jagadeesan, Dexter Kozen, Paul B. Levy, and the anonymous reviewers for fruitful discussions and suggestions.

## REFERENCES

R. J. Aumann. 1961. Borel structures for function spaces. *Illinois Journal of Mathematics* 5 (1961), 614–630.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2015. A Lambda-Calculus Foundation for Universal Probabilistic Programming. *CoRR* abs/1512.08990 (2015). http://arxiv.org/abs/1512.08990

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 33–46.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles* 76, 1 (2017), 1–32. https://doi.org/10.18637/jss.v076.i01

Arnaud Doucet and Adam M. Johansen. 2011. A Tutorial on Particle Filtering and Smoothing: Fifteen years later. In *The Oxford Handbook of Nonlinear Filtering*, Dan Crisan and Boris Rozovskii (Eds.). Oxford University Press, 656–704.

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75.

Marcelo Fiore and Philip Saville. 2017. List Objects with Algebraic Structure. In *2st International Conference on Formal Structures for Computation and Deduction, FSCD 2017*.

Herman Geuvers and Erik Poll. 2007. Iteration and primitive recursion in categorical terms. In *Reflections on Type Theory, Lambda Calculus, and the Mind, Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday*. Radboud Universiteit, Nijmegen, 101–114.

Charles J. Geyer. 2011. Introduction to Markov Chain Monte Carlo. In *Handbook of Markov Chain Monte Carlo*, Steve Brooks, Andrew Gelman, Galin L. Jones, and Xiao-Li Meng (Eds.). Chapman and Hall/CRC, Chapter 1, 3–48.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2008. Church: a language for generative models. In *UAI*.

Noah Goodman and Andreas Stuhlmüller. 2014. Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014).

Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. 2014. Tabular: A Schema-driven Probabilistic Programming Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 321–334. https://doi.org/10.1145/2535838.2535850

Esfandiar Haghverdi and Philip Scott. 2006. A categorical model for the geometry of interaction. *Theoretical Computer Science* 350, 2 (2006), 252 – 274. https://doi.org/10.1016/j.tcs.2005.10.028 Automata, Languages and Programming: Logic and Semantics (ICALP-B 2004).

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-Order Probability Theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '17, Reykjavik, Iceland, June 20-23, 2017*.

Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. 2015. A Provably Correct Sampler for Probabilistic Programs. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*. 475–488.

Graham Hutton. 1999. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.* 9, 4 (July 1999), 355–372. https://doi.org/10.1017/S0956796899003500

Bart Jacobs. 2017. From Probability Monads to Commutative Effectuses. *Journ. of Logical and Algebraic Methods in Programming* (2017). To appear.

Mauro Jaskelioff. 2009. *Lifting of Operations in Modular Monadic Semantics*. Ph.D. Dissertation. University of Nottingham.

G M Kelly. 1980. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. *Bull. Austral. Math. Soc.* 22 (1980), 1–83.

Anders Kock. 1972. Strong functors and monoidal monads. *Archiv der Mathematik* 23, 1 (1972), 113–120.

Anders Kock. 2012. Commutative monads as a theory of distributions. *Theory and Applications of Categories* 26, 4 (2012), 97–131.

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David Blei. 2015. Automatic Variational Inference in Stan. In *NIPS*. https://papers.nips.cc/paper/5758-automatic-variational-inference-in-stan

Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. 2017. Inference Compilation and Universal Probabilistic Programming. In *AISTATS*. http://www.tuananhle.co.uk/assets/pdf/le2016inference.pdf

Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv:1404.0099* (2014). http://arxiv.org/abs/1404.0099

Francisco Marmolejo and Richard J. Wood. 2010. Monads as extension systems — no iteration is necessary. *Theory and Applications of Categories* 24, 4 (2010), 84–113.

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. (2014). Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *LICS*. IEEE Computer Society, USA, 14–23.

Lawrence M. Murray. 2013. Bayesian State-Space Modelling on High-Performance Hardware Using LibBi. arXiv:1306.3277. (2013).

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. 2005. A probabilistic language based upon sampling functions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 171–182.

Maciej Piróg. 2016. Eilenberg-Moore Monoids and Backtracking Monad Transformers. In *Proceedings 6th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2016, Eindhoven, Netherlands, 8th April 2016. (EPTCS)*, Robert Atkey and Neelakantan R. Krishnaswami (Eds.), Vol. 207. 23–56. https://doi.org/10.4204/EPTCS.207.2

Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. 154–165.

Adam Ścibior, Zoubin Ghahramani, and Andrew Gordon. 2015. Practical Probabilistic Programming with Monads. In *Haskell*. http://dl.acm.org/citation.cfm?id=2804317

Sam Staton. 2017. Commutative semantics for probabilistic programming. In *Proc. ESOP 2017*.

Andrew Thomas, David J. Spiegelhalter, and W. R. Gilks. 1992. BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian statistics* 4 (1992), 837–842. Issue 9.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *ICLR*.

David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *AISTATS*. https://web.stanford.edu/~ngoodman/papers/lightweight-mcmc-aistats2011.pdf The published version contains a serious bug in the definition of alpha that was fixed in revision 3 available at the given URL.

David Wingate and Theophane Weber. 2013. Automated Variational Inference in Probabilistic Programming. arXiv:1301.1299. (2013).

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*. 1024–1032.

Robert Zinkov and Chung-chieh Shan. 2017. Composing inference algorithms as program transformations. In *Proceedings of Uncertainty in Artificial Intelligence*.